REST APIs - REST Web Services

You will learn how to …

- Create REST APIs / Web Services with Spring
- Discuss REST concepts, JSON and HTTP messaging
- Install REST client tool: Postman
- Develop REST APIs / Web Services with **@RestController**
- Build a CRUD interface to the database with Spring REST

Practical Results

- Introduction to Spring REST development

- Not an A to Z reference … for that you can see Spring Reference Manual

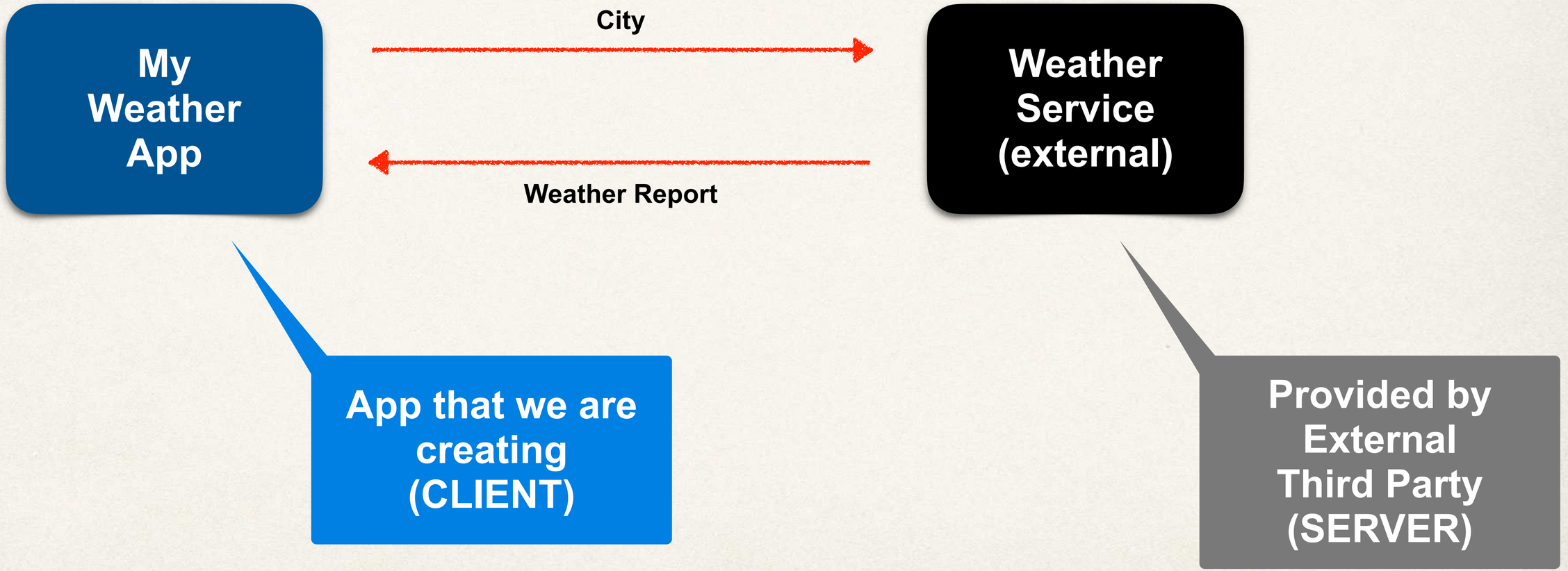
https://projects.spring.io/spring-framework/

Business Problem

- Build a client app that provides the weather report for a city

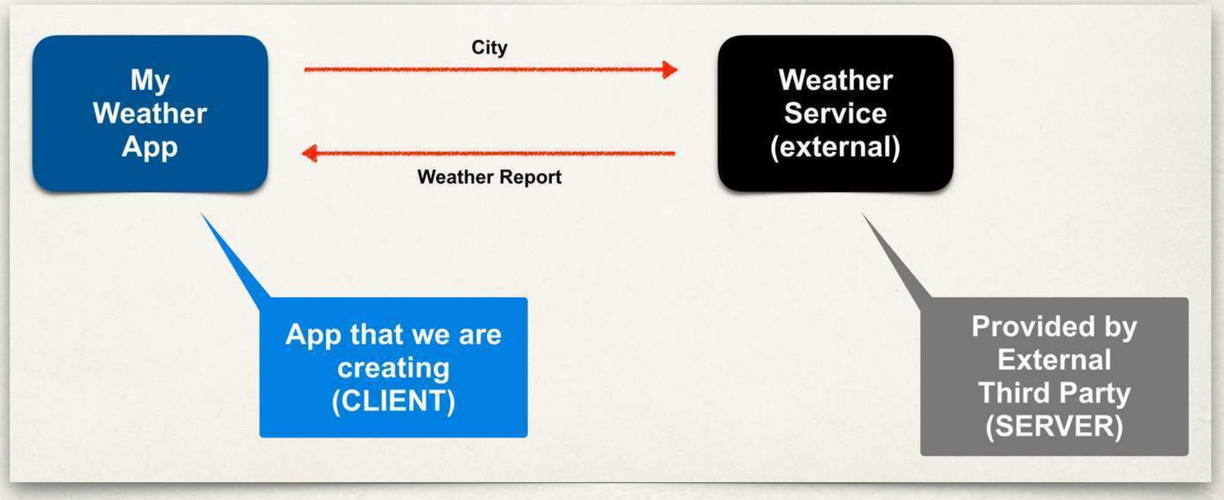
- Need to get weather data from an external service

Application Architecture



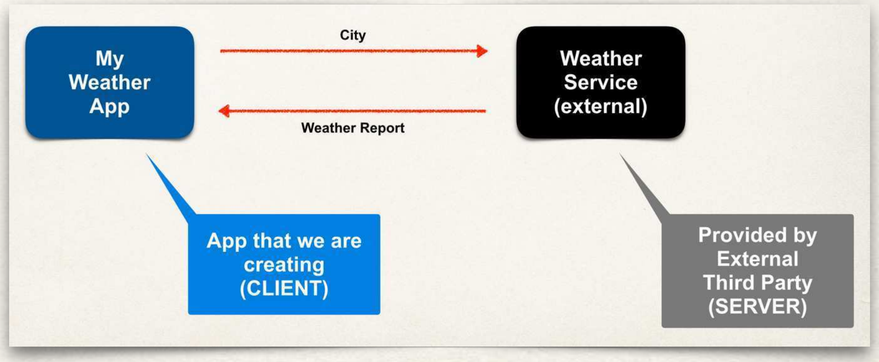
Questions

- How will we connect to the Weather Service?
- What programming language do we use?
- What is the data format?



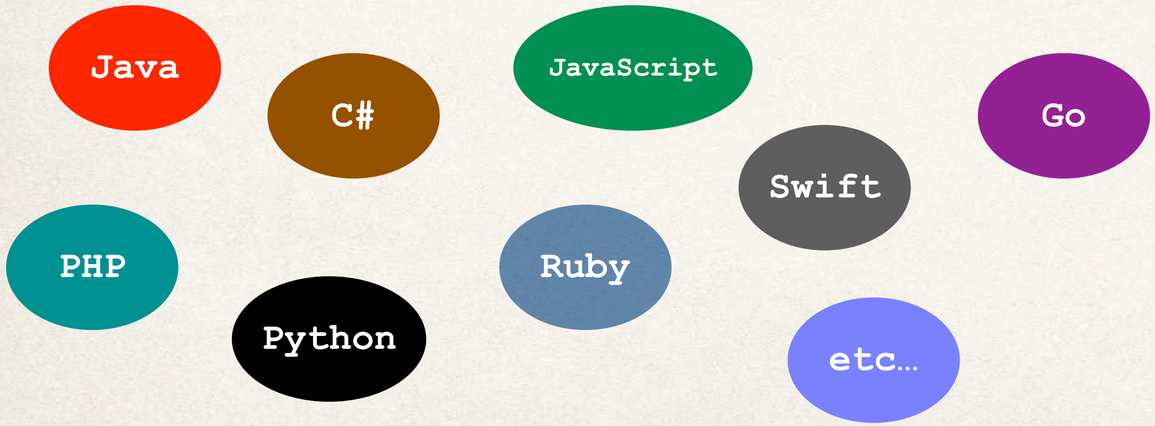
Answers

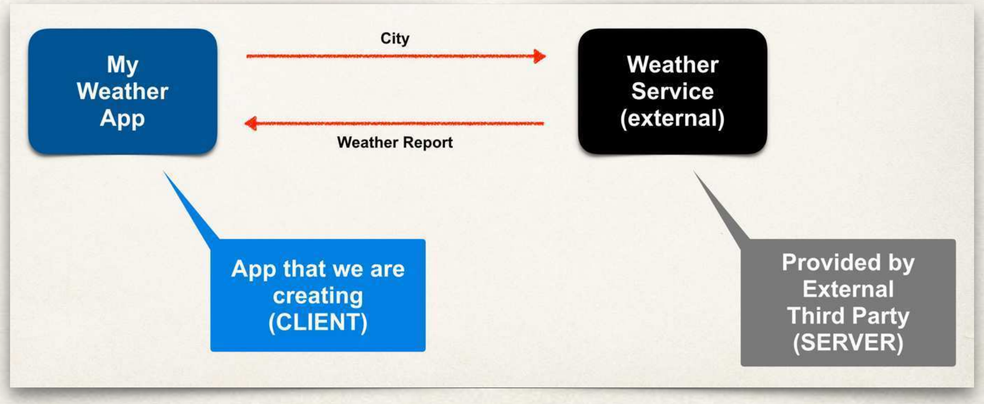
- **How will we connect to the Weather Service?**
  - We can make REST API calls over HTTP
  - REST: REpresentational State Transfer
  - Lightweight approach for communicating between applications



Answers

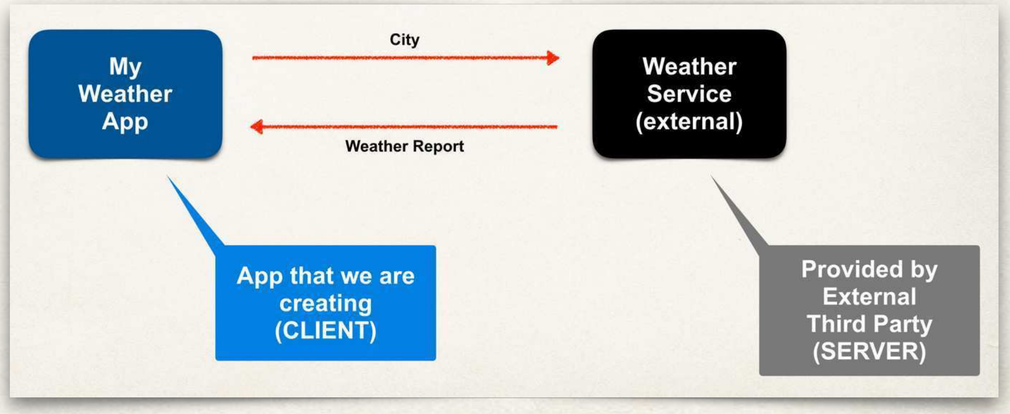
- **What programming language do we use?**
  - REST is language independent
  - The **client** application can use **ANY** programming language
  - The **server** application can use **ANY** programming language





Answers

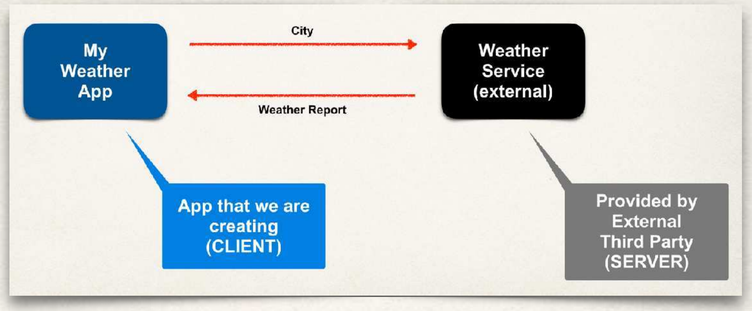
- **What is the data format?**
  - REST applications can use any data format
  - Commonly see XML and JSON
  - JSON is most popular and modern
    - JavaScript Object Notation



Possible Solution

- Use online Weather Service API provided by: openweathermap.org
- Provide weather data via an API
- Data is available in multiple formats: JSON, XML etc …

Call Weather Service



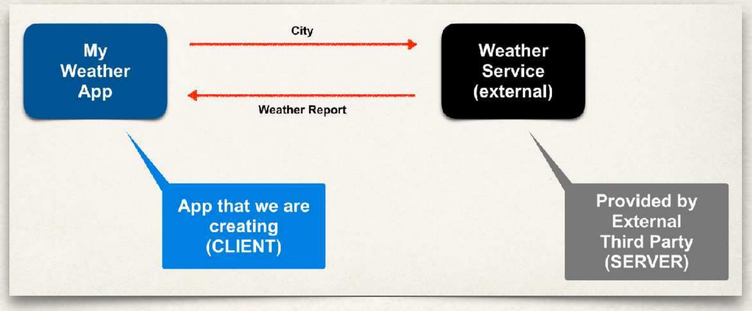
- The API documentation gives us the following:
  - Pass in the latitude and longitude for your desired location

```
api.openweathermap.org/data/<apiVersion>/onecall?lat={theLatitude}&lon={theLongitude}
```

Pass the latitude and longitude of the city

Response - Weather Report

- The Weather Service responds with JSON

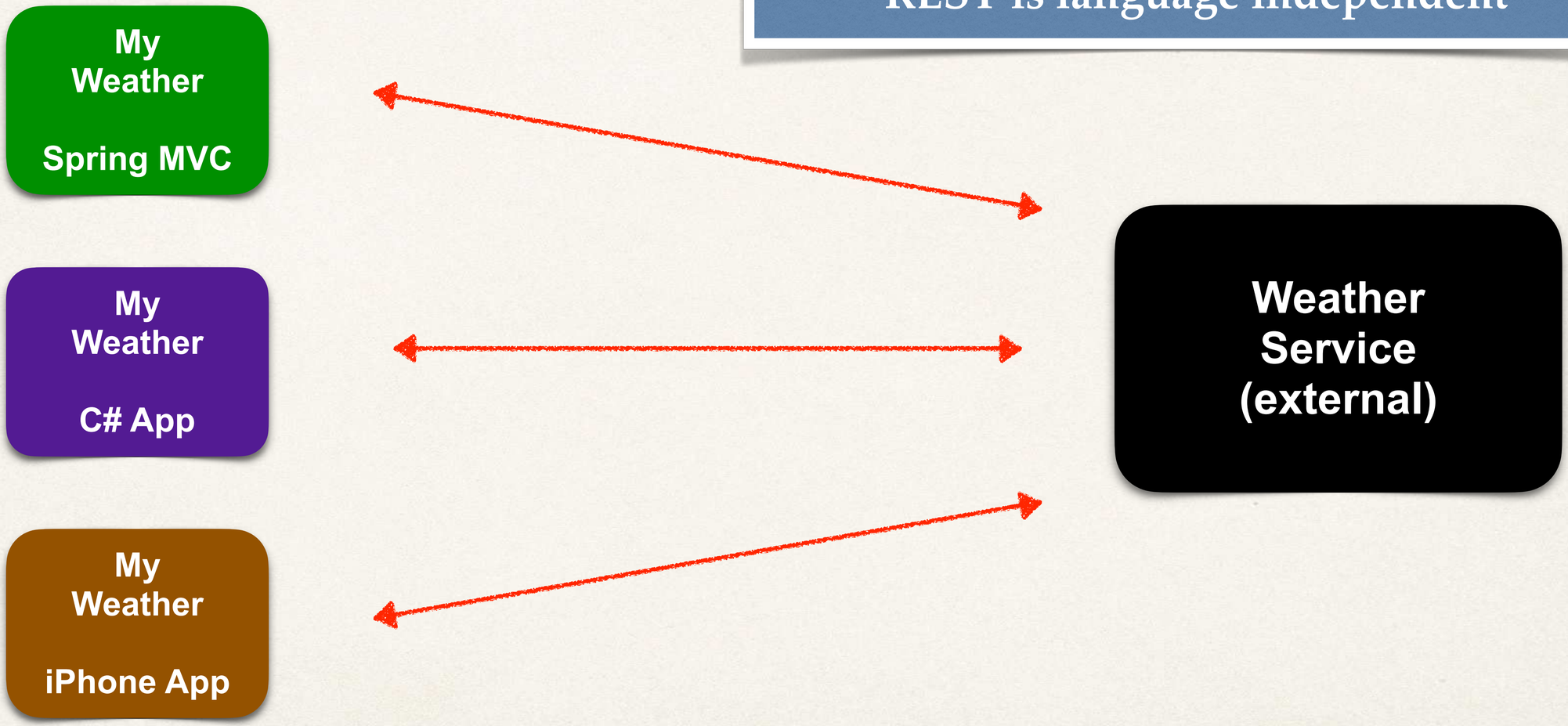



```
{
   …
   "temp": xxx,
   "feels_like": yyy,
   "humidity": zzz,
   …
}
```

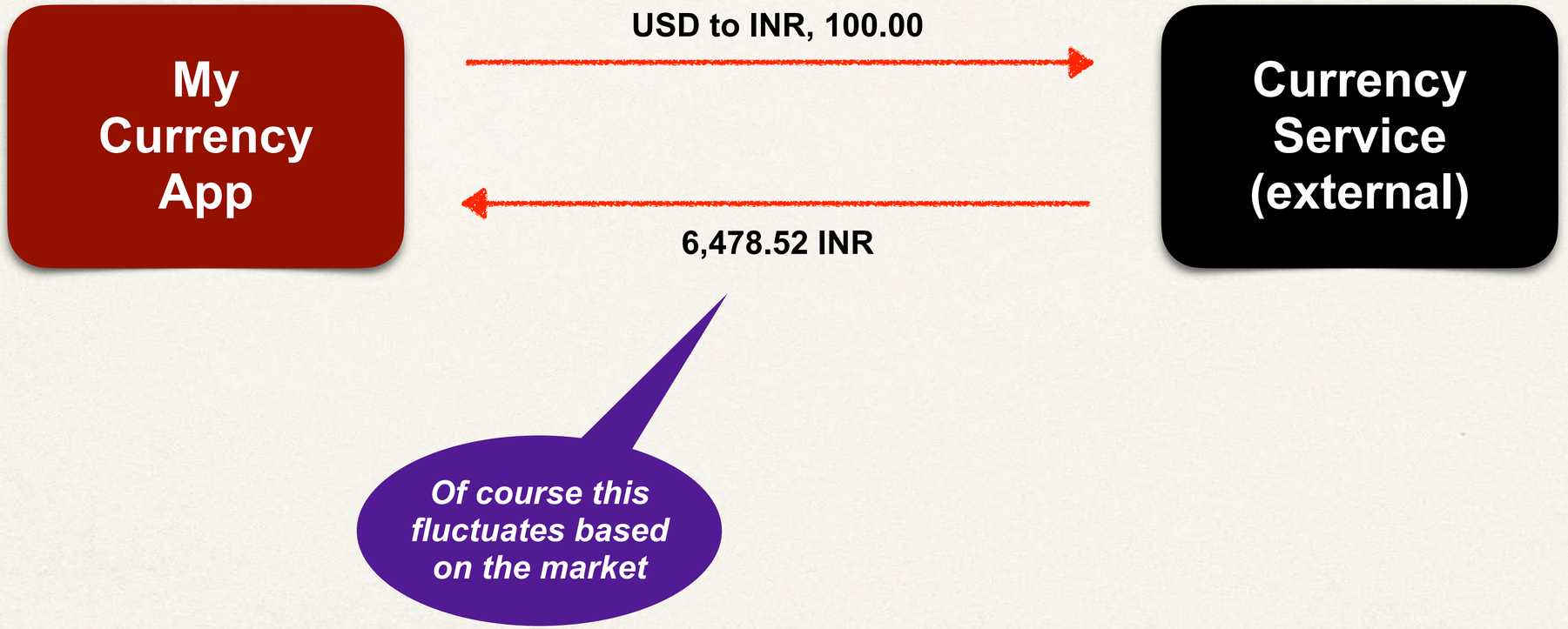
Condensed version

Multiple Client Apps

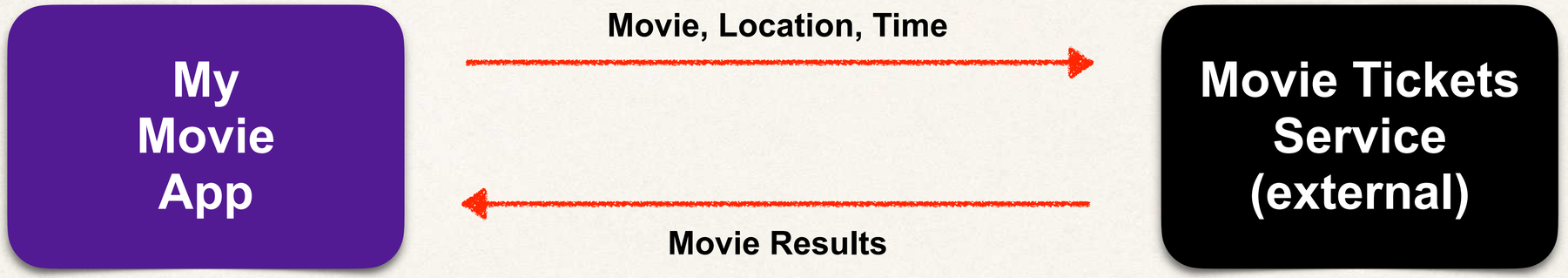
Remember:
REST calls can be made over HTTP
REST is language independent



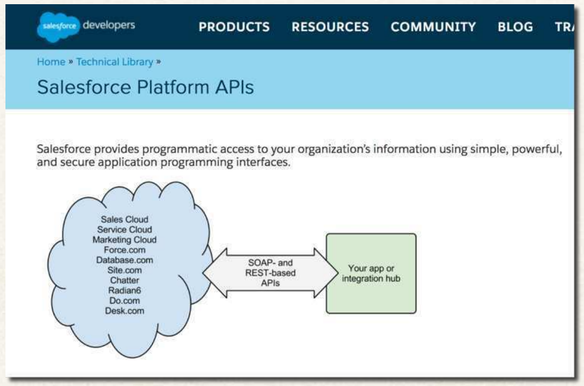
Currency Converter App



Movie Tickets App



Customer Relationship Manager (CRM) App



We will create this code using Spring REST (SERVER)

What do we call it?

REST API

REST
Web Services

REST Services

RESTful API

RESTful
Web Services

RESTful Services

Generally, all mean the SAME thing

What do we call it?

REST API

REST
Web Services

REST Services

RESTful API

RESTful
Web Services

RESTful Services

Generally, all mean the SAME thing

JSON Basics - Syntax

What is JSON?

- **J**ava**S**cript **O**bject **N**otation

- Lightweight data format for storing and exchanging data … plain text

- Language independent … not just for **JavaScript**

JSON is just plain text data

- Can use with any programming language: **Java, C# , Python** etc …

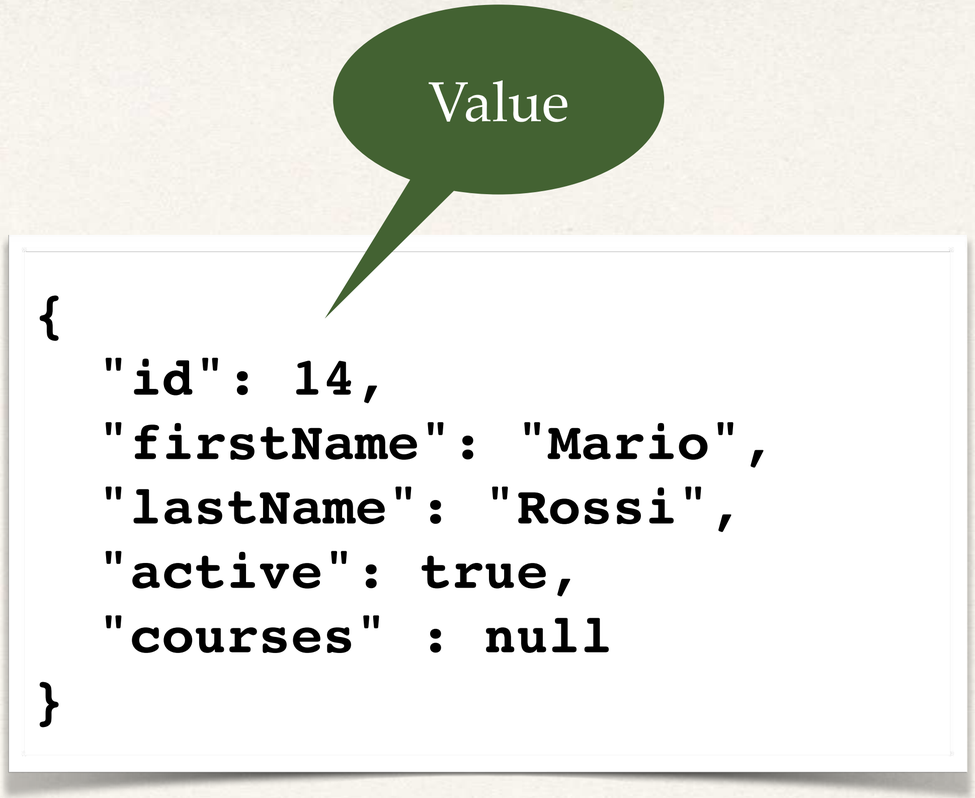
Simple JSON Example

- Curley braces define objects in JSON
- Object members are name / value pairs
  - Delimited by colons
- Name is **always** in double-quotes


```
{
   "id": 14,
   "firstName": "Mario",
   "lastName": "Rossi",
   "active": true
}
```

JSON Values

- Numbers: no quotes
- String: in double quotes
- Boolean: **`true`**, **`false`**
- Nested JSON object
- Array
- **`null`**



Nested JSON Objects

```
{
   "id": 14,
   "firstName": "Mario",
   "lastName": "Rossi",
   "active": true,
   "address" : {
                  "street" : "100 Main St",
                  "city" : "Philadelphia",
                  "state" : "Pennsylvania",
                  "zip" : "19103",
                  "country" : "USA"
               }
}
```

JSON Arrays

```
{
  "id": 14,
  "firstName": "Mario",
  "lastName": "Rossi",
  "active": true,
  "languages" : ["Java", "C#", "Python", "Javascript"]
}
```

REST HTTP Basics

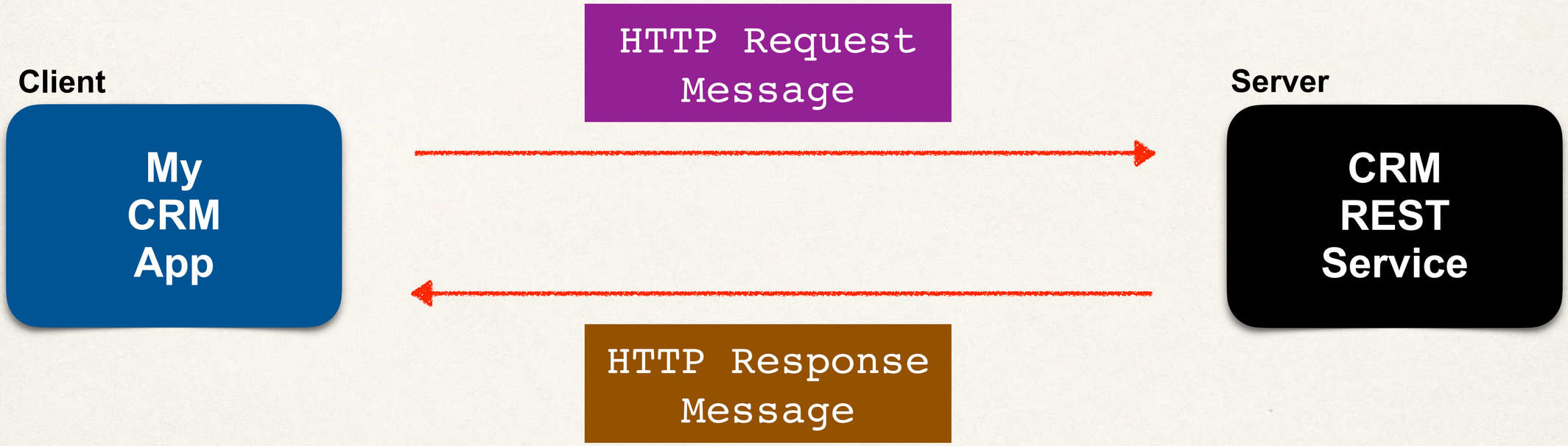
REST over HTTP

- Most common use of REST is over HTTP

- Leverage HTTP methods for CRUD operations

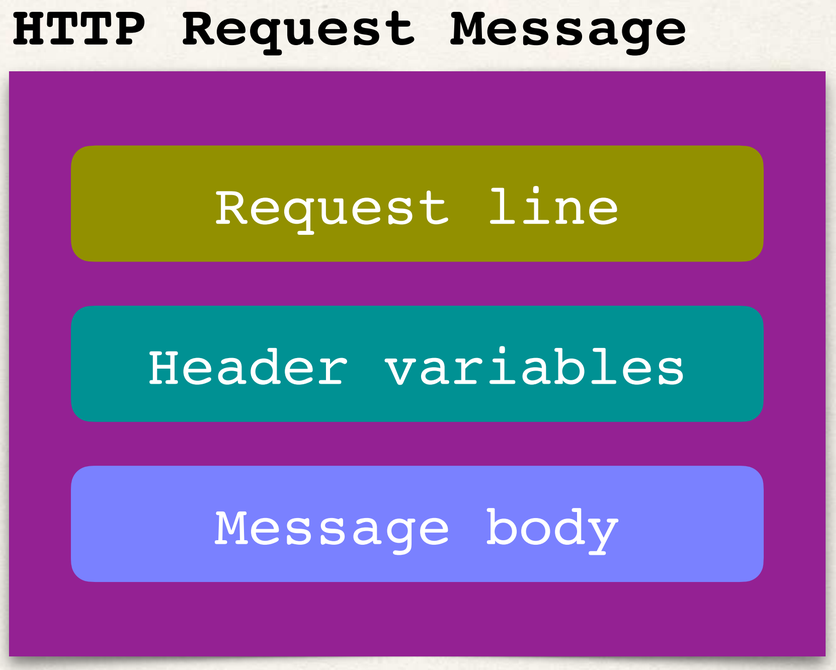
HTTP Method	CRUD Operation
POST	Create a new entity
GET	Read a list of entities or single entity
PUT	Update an existing entity
DELETE	Delete an existing entity

HTTP Messages



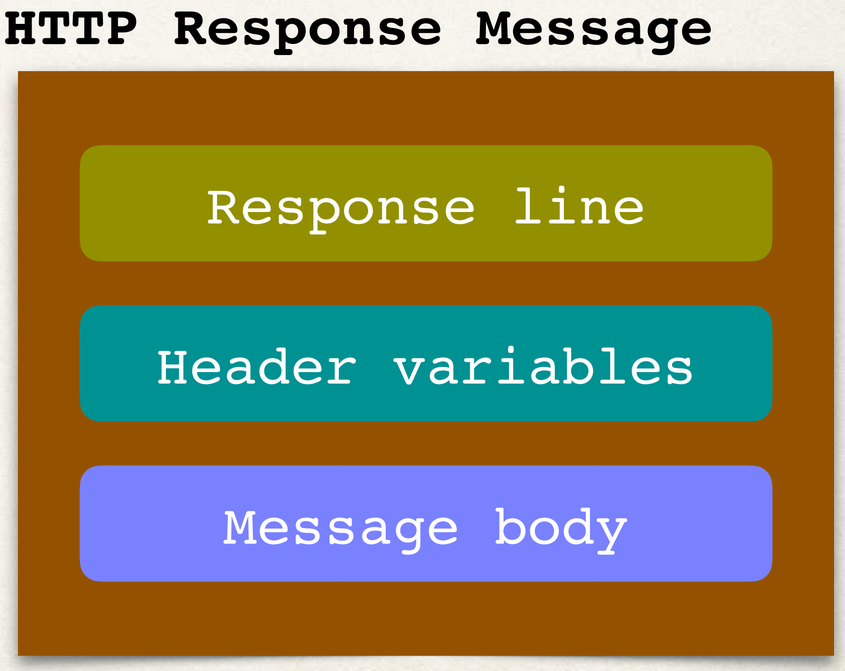
HTTP Request Message

- Request line: the HTTP command
- Header variables: request metadata
- Message body: contents of message



HTTP Response Message

- Response line: server protocol and status code
- Header variables: response metadata
- Message body: contents of message



HTTP Response - Status Codes

Code Range	Description
100 - 199	Informational
200 - 299	Successful
300 - 399	Redirection
400 - 499	Client error
500 - 599	Server error

401 Authentication Required
404 File Not Found

500 Internal Server Error

MIME Content Types

- The message format is described by MIME content type
  - **M**ultipurpose **I**nternet **M**ail-**E**xtension
- Basic Syntax: `type/sub-type`
- Examples
  - `text/html`, `text/plain`
  - `application/json`, `application/xml`, ...

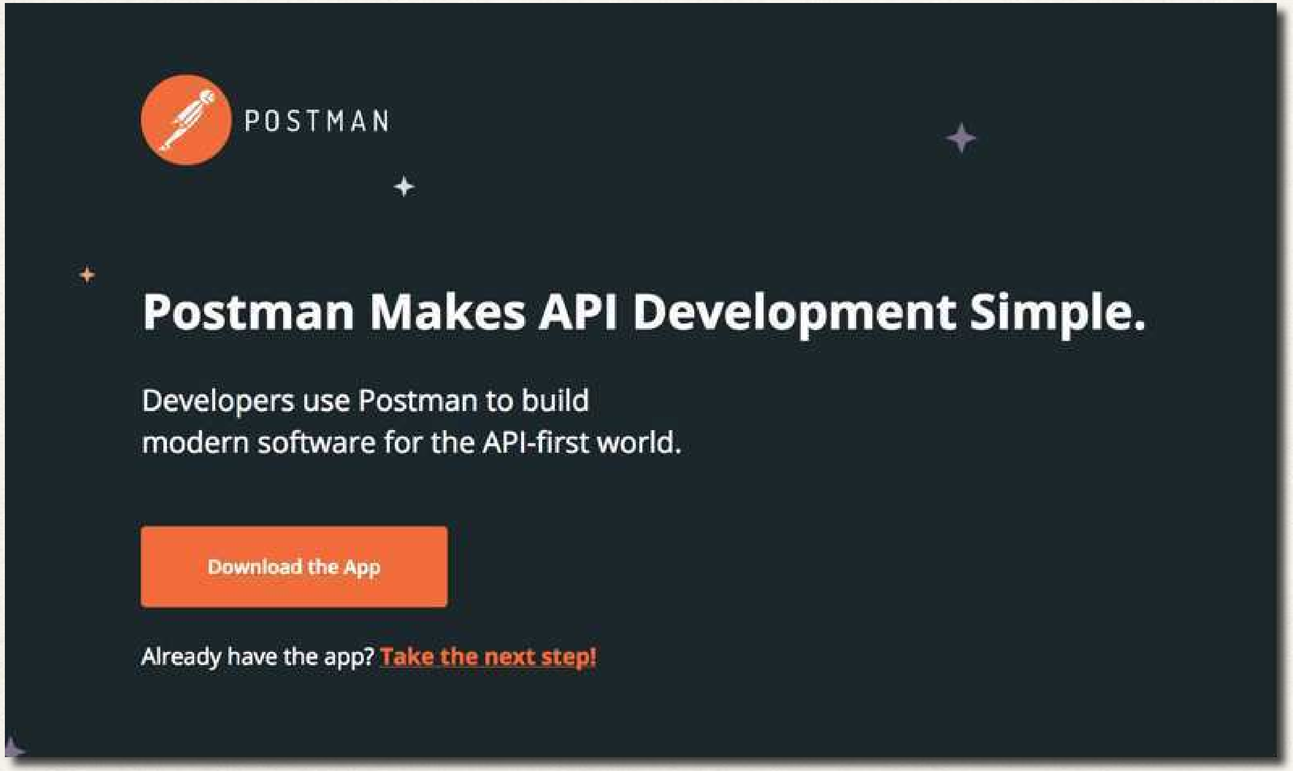
Client Tool

- We need a **client tool**

- Send HTTP requests to the REST Web Service / API

- Plenty of tools available: curl, Postman, etc …

Postman

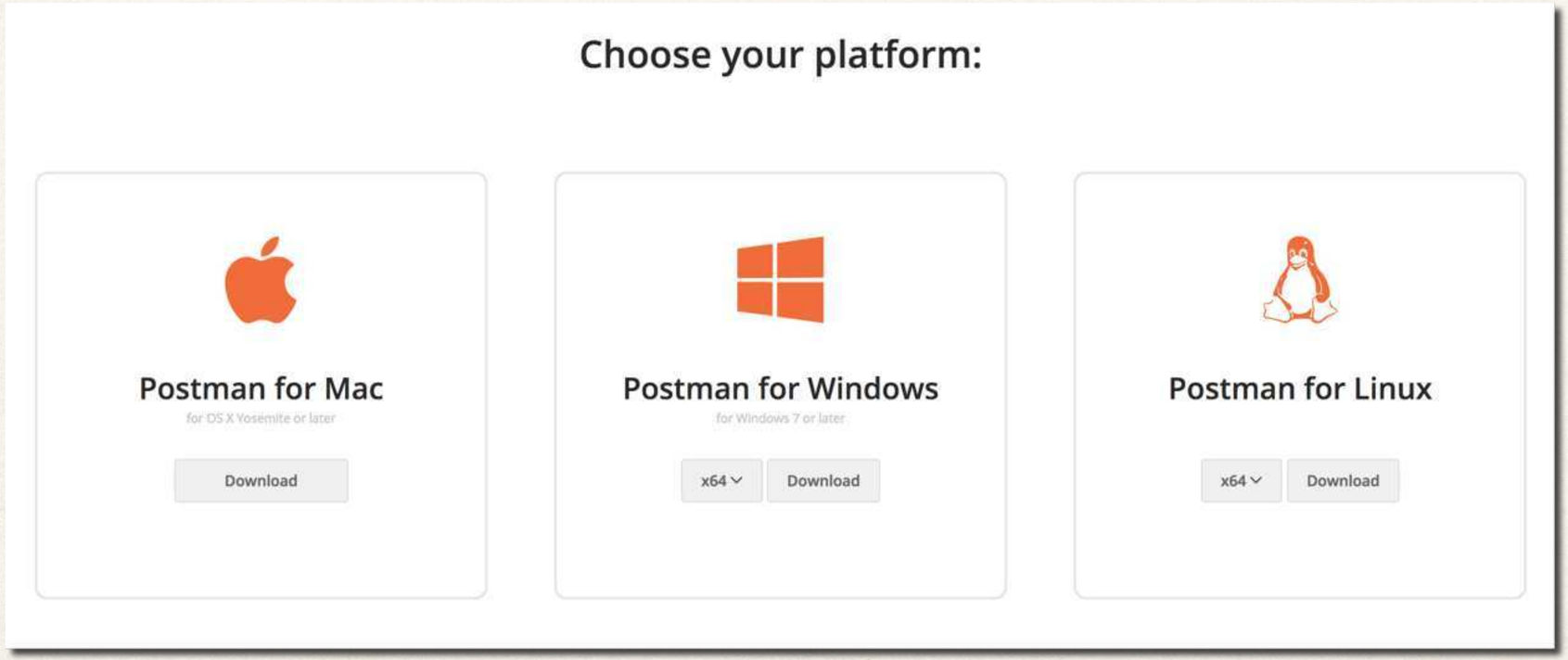


Free developer plan

www.getpostman.com

Install Postman Now

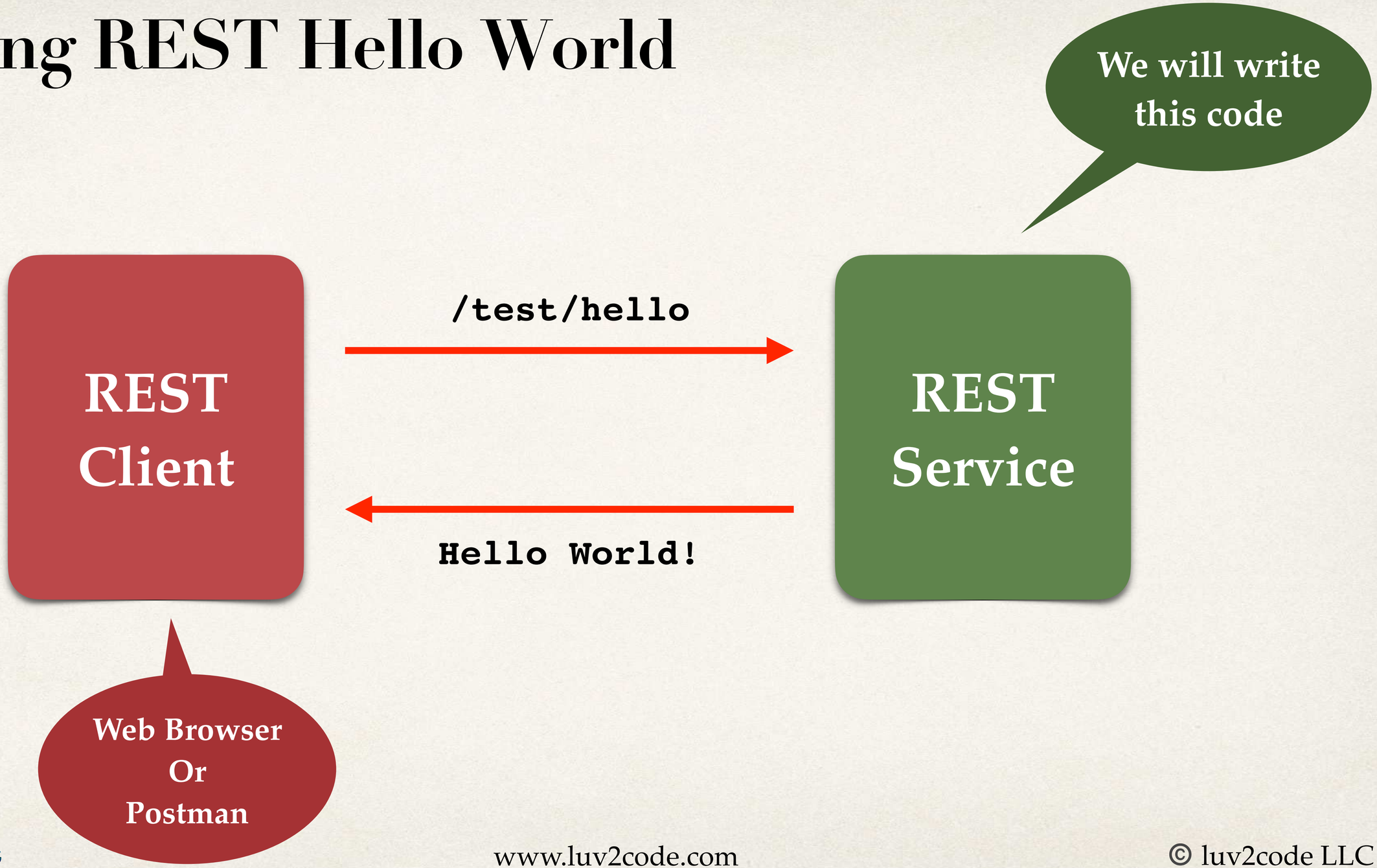
www.getpostman.com



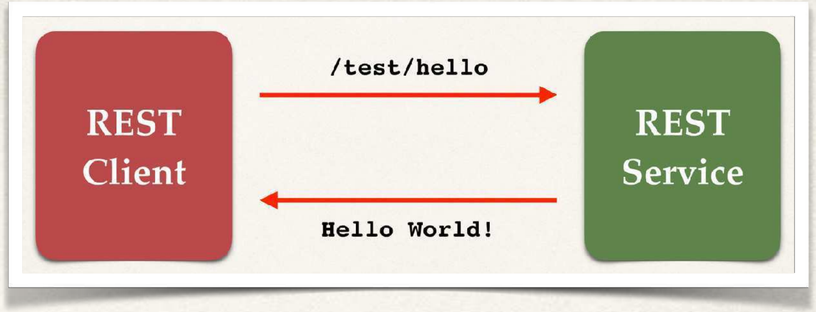
Postman Demo

Spring REST Controller

Spring REST Hello World



Spring REST Controller



Adds REST support

```
@RestController
@RequestMapping("/test")
public class DemoRestController {

  @GetMapping("/hello")
  public String sayHello() {
    return "Hello World!";
  }

}
```

Access the REST endpoint at `/test/hello`

Returns content to client

Testing with REST Client - Postman

Testing with REST Client - Web Browser

Access the REST endpoint at `/test/hello`

The response

Web Browser vs Postman

- For simple REST testing for GET requests
  - Web Browser and Postman are similar

- However, for advanced REST testing: POST, PUT etc …
  - Postman has much better support
  - POSTing JSON data, setting content type
  - Passing HTTP request headers, authentication etc …

Spring REST Controller - Dev Process

Development Process

Step-By-Step

1. Add Maven dependency for Spring Boot Starter Web

2.Create Spring REST Service using @RestController

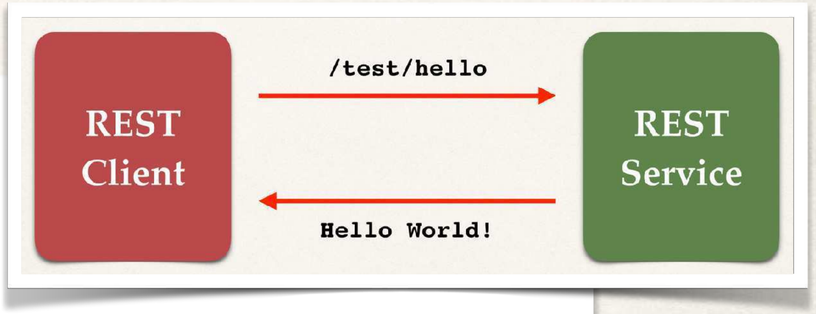
Step 1: Add Maven Dependency

File: pom.xml

```
<!-- Add Spring Boot Starter Web -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

**At Spring Initializr website,
can also select the “Web” dependency**

Step 2: Create Spring REST Service



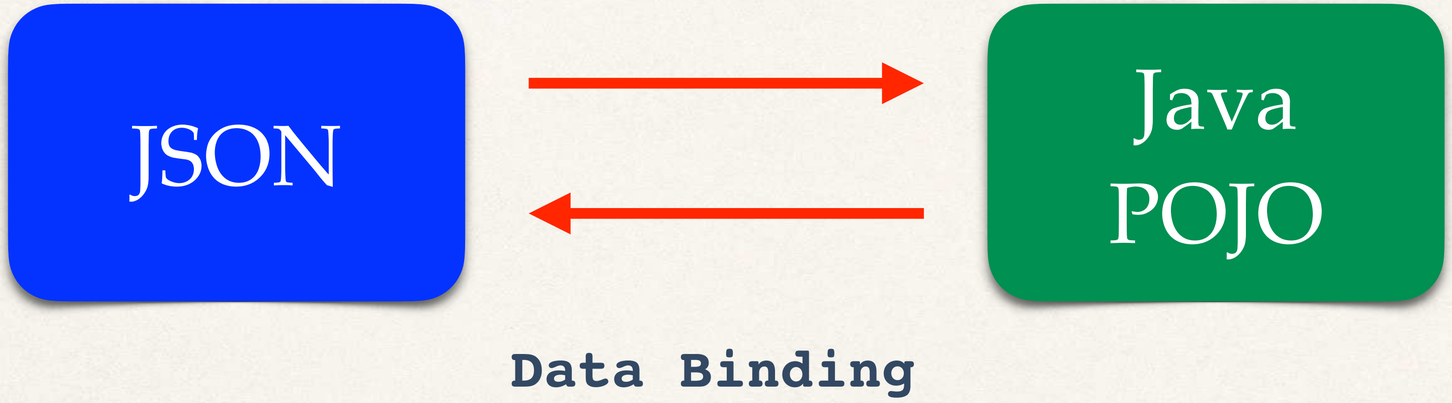

```
@RestController
@RequestMapping("/test")
public class DemoRestController {

  @GetMapping("/hello")
  public String sayHello() {
    return "Hello World!";
  }

}
```

Handles HTTP GET requests

Java JSON Data Binding

Java JSON Data Binding

- Data binding is the process of converting JSON data to a Java POJO



Also known as

Mapping

Serialization / Deserialization

Marshalling / Unmarshalling

JSON Data Binding with Jackson

- Spring uses the **Jackson Project** behind the scenes

- Jackson handles data binding between JSON and Java POJO

- Details on Jackson Project:

https://github.com/FasterXML/jackson-databind

Jackson Data Binding

- By default, Jackson will call appropriate getter/setter method

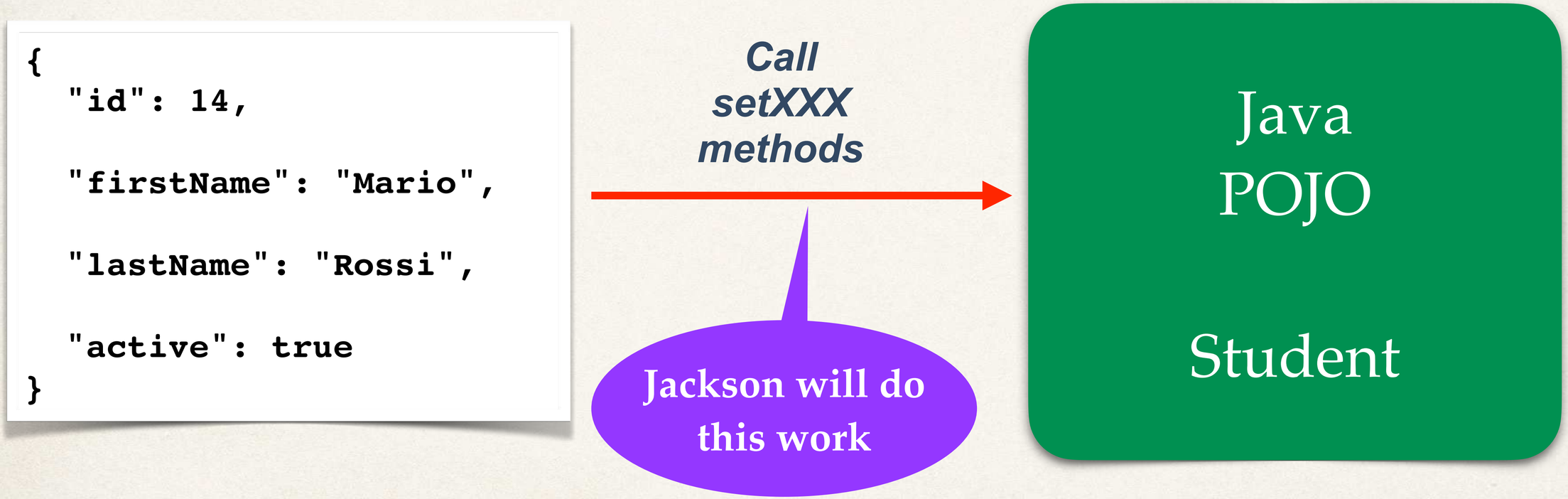
```
{
  "id": 14,

  "firstName": "Mario",

  "lastName": "Rossi",

  "active": true
}
```

Java
POJO

Student

JSON to Java POJO

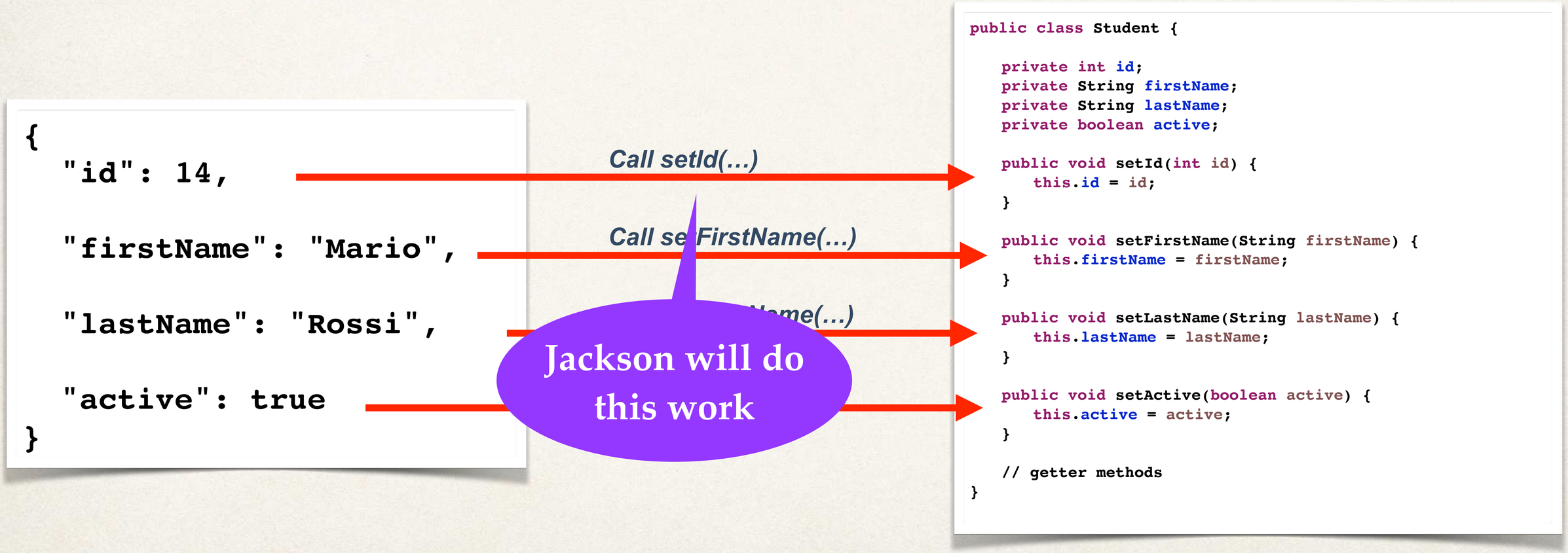
- Convert JSON to Java POJO … call setter methods on POJO



JSON to Java POJO

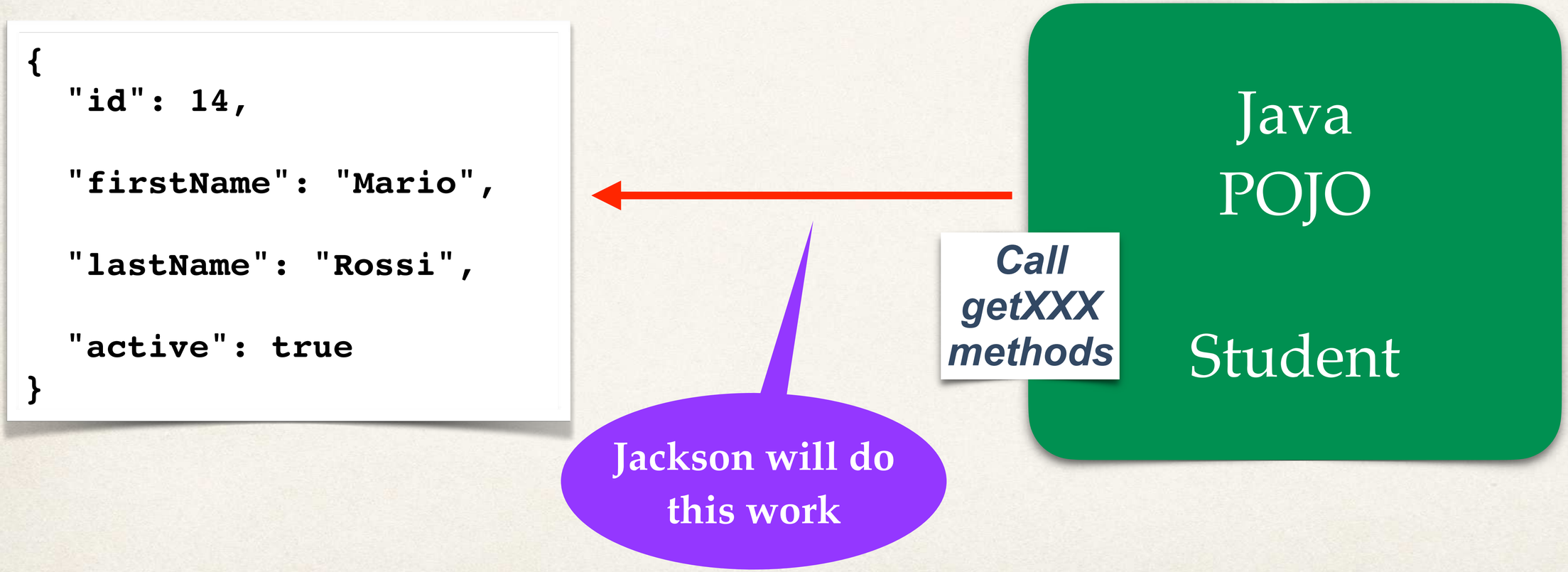
Note: Jackson calls the setXXX methods
It does NOT access internal private fields directly

- Convert JSON to Java POJO … call setter methods on POJO



Java POJO to JSON

- Now, let's go the other direction
- Convert Java POJO to JSON … call getter methods on POJO



Spring and Jackson Support

- When building Spring REST applications
- Spring will automatically handle Jackson Integration
- JSON data being passed to REST controller is converted to POJO
- Java object being returned from REST controller is converted to JSON

Happens automatically behind the scenes

Spring REST Service - Students

Create a New Service

- Return a list of students

`/api/students` *Returns a list of students*

Spring REST Service

We will write this code

REST Client

/api/students

REST Service

Web Browser Or Postman

Convert Java POJO to JSON

- Our REST Service will return **`List<Student>`**

- Need to convert **`List<Student>`** to JSON

- **Jackson** can help us out with this …

Spring Boot and Jackson Support

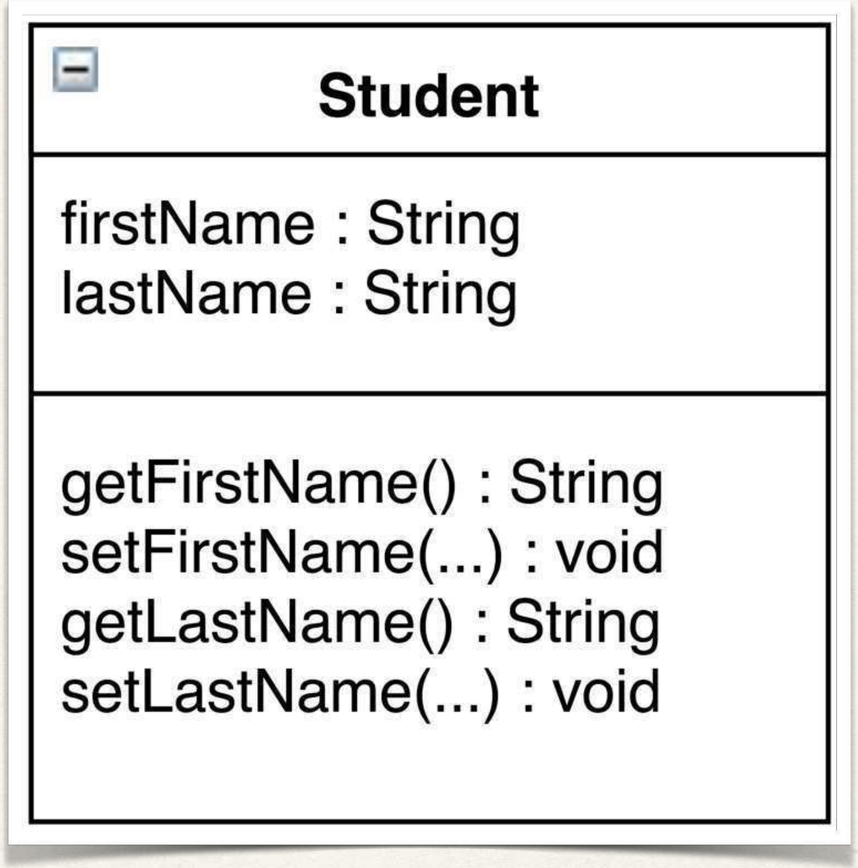
Happens automatically behind the scenes

- Spring Boot will automatically handle **Jackson** integration
- JSON data being passed to REST controller is converted to Java POJO
- Java POJO being returned from REST controller is converted to JSON

Student POJO (class)

Java
POJO

Student

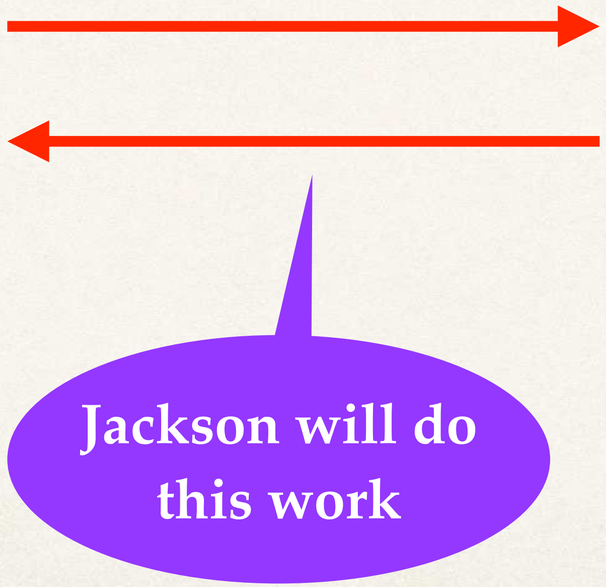


Jackson Data Binding

- Jackson will call appropriate getter / setter method

Remember

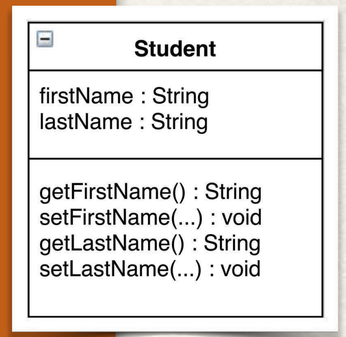
```
{
  "id": 14,

  "firstName": "Mario",

  "lastName": "Rossi",

  "active": true
}
```



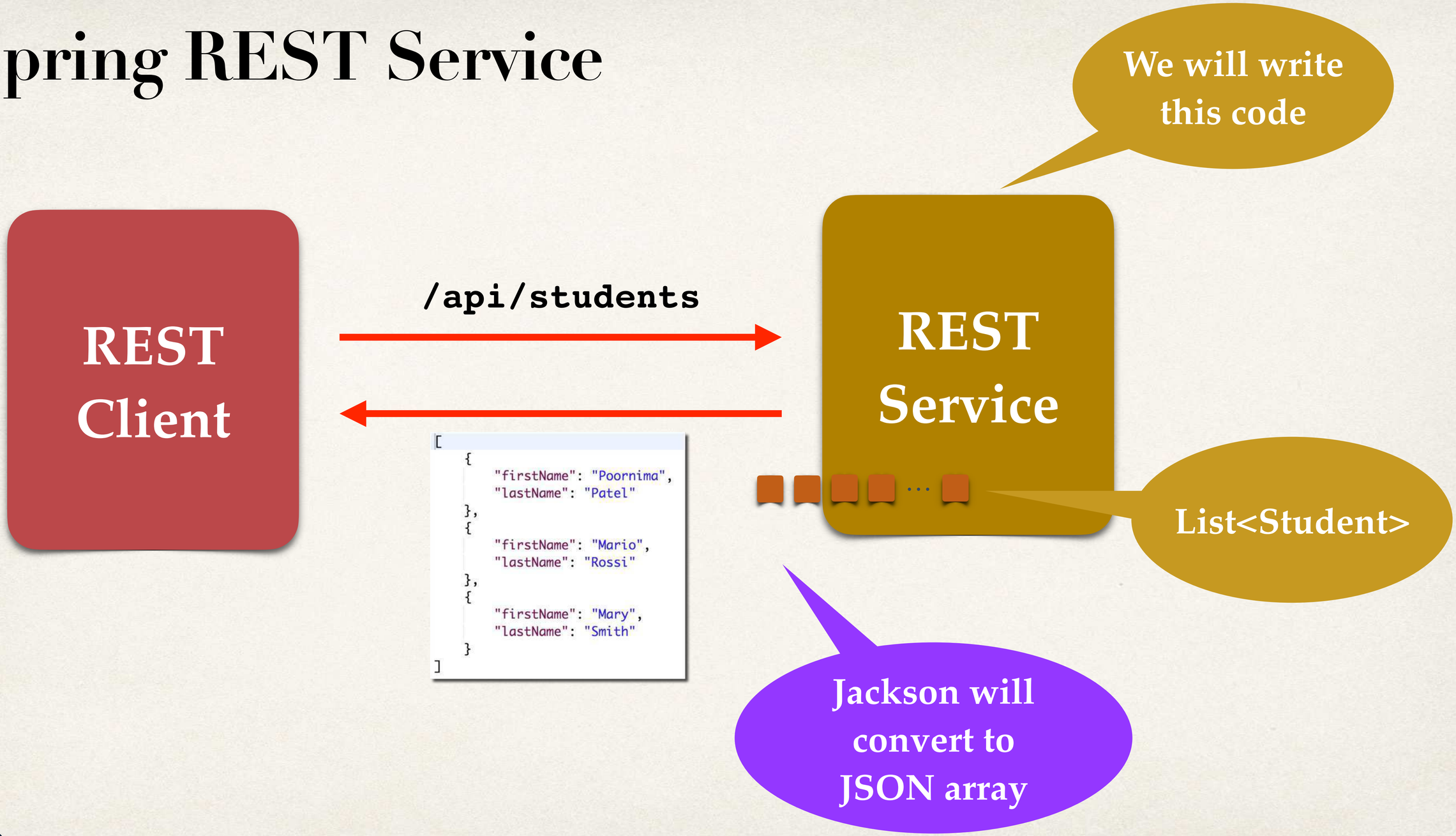
Jackson will do this work

Java POJO

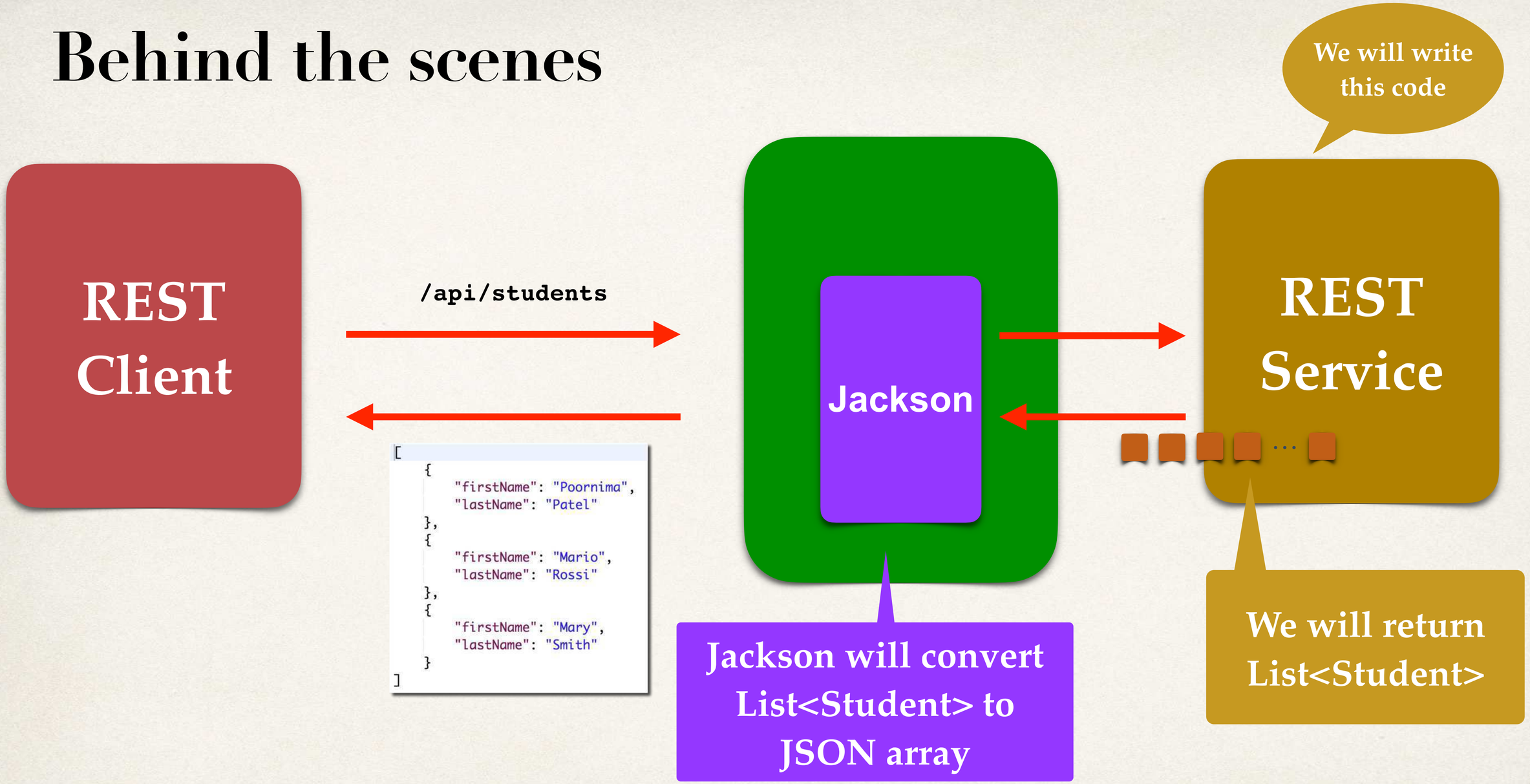
Student



Spring REST Service



Behind the scenes



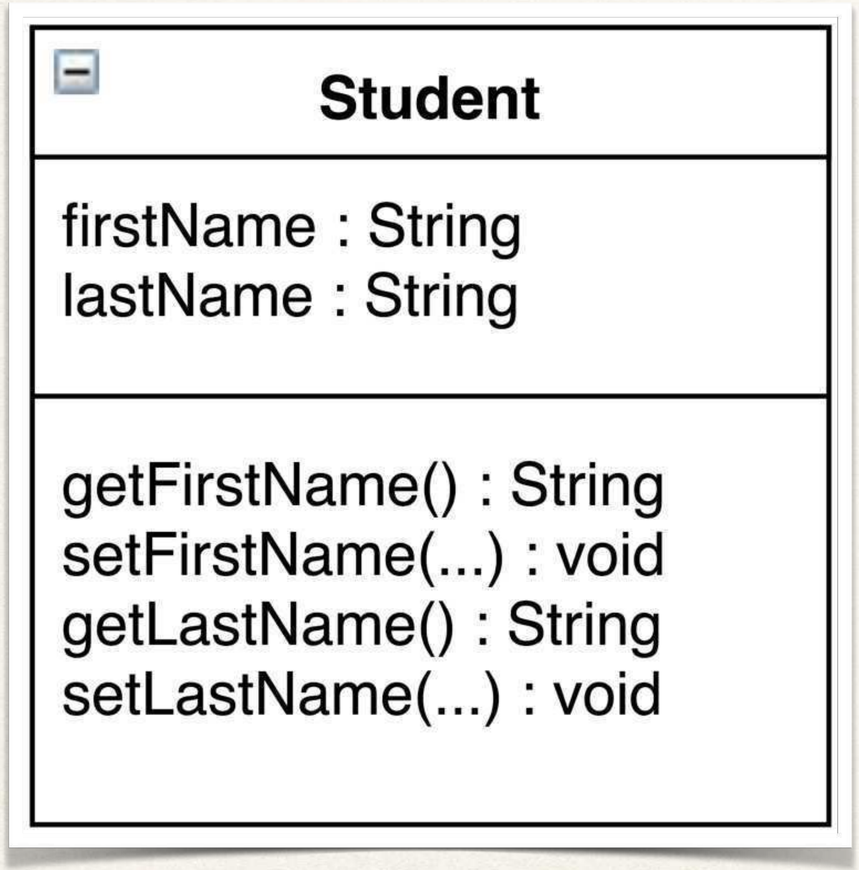
Development Process

Step-By-Step

1. Create Java POJO class for **Student**

2.Create Spring REST Service using **@RestController**

Step 1: Create Java POJO class for Student



File: Student.java

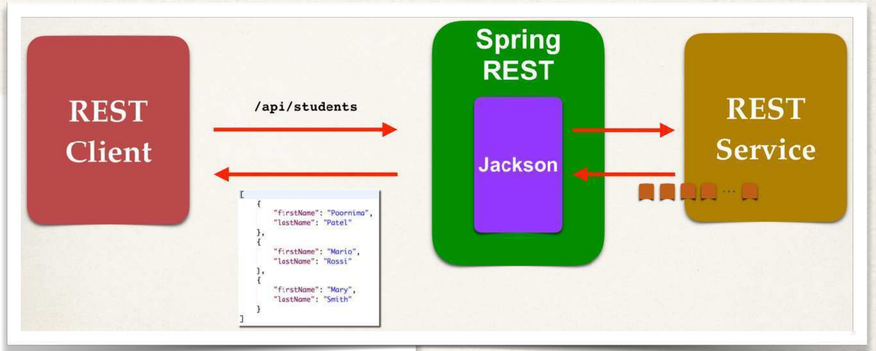
```
public class Student {

    private String firstName;
    private String lastName;

    public Student() {

    }

    public Student(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

}
```

Fields
Constructors
Getter/Setters

Step 2: Create @RestController

File: StudentRestController.java

```
@RestController
@RequestMapping("/api")
public class StudentRestController {

   // define endpoint for "/students" - return list of students

   @GetMapping("/students")
   public List<Student> getStudents() {

      List<Student> theStudents = new ArrayList<>();

      theStudents.add(new Student("Poornima", "Patel"));
      theStudents.add(new Student("Mario", "Rossi"));
      theStudents.add(new Student("Mary", "Smith"));

      return theStudents;
   }

}
```



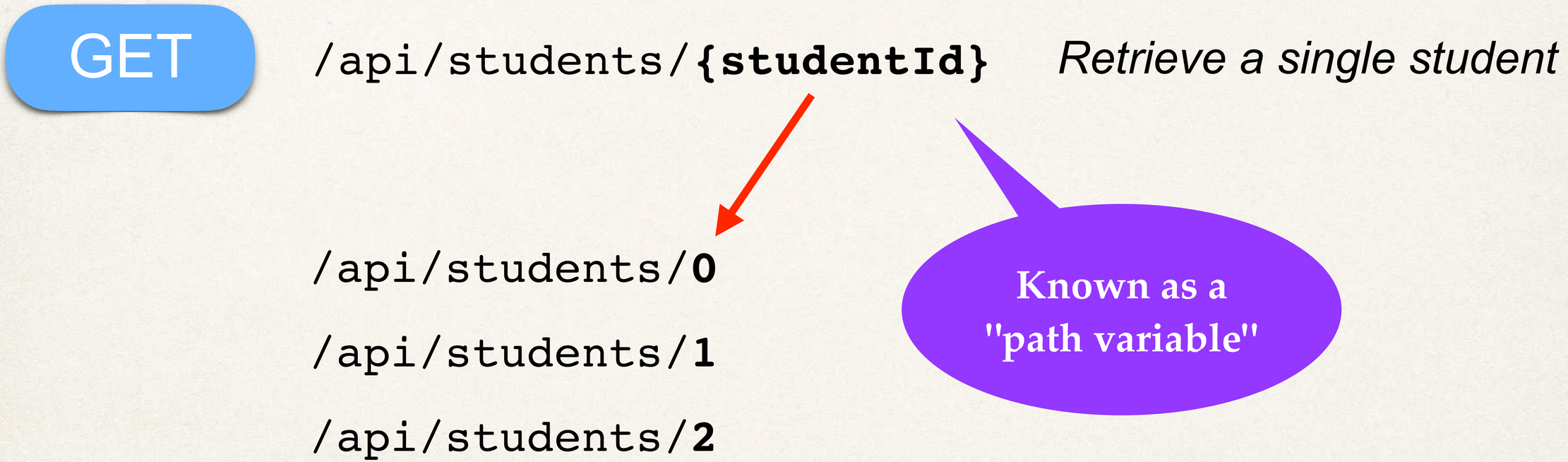
We'll hard code for now … can add DB later …

Jackson will convert List<Student> to JSON array

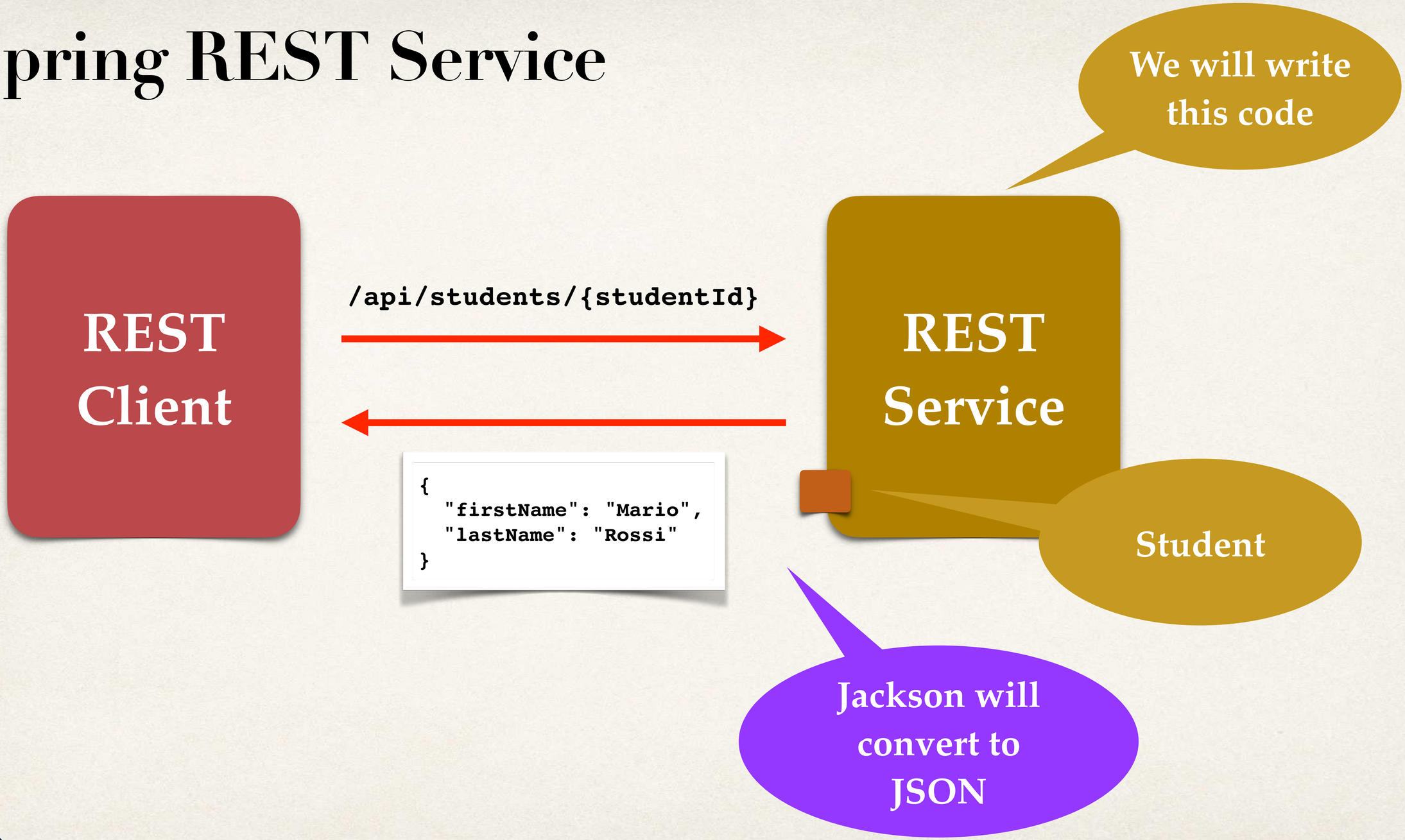
Spring REST - Path Variables

Path Variables

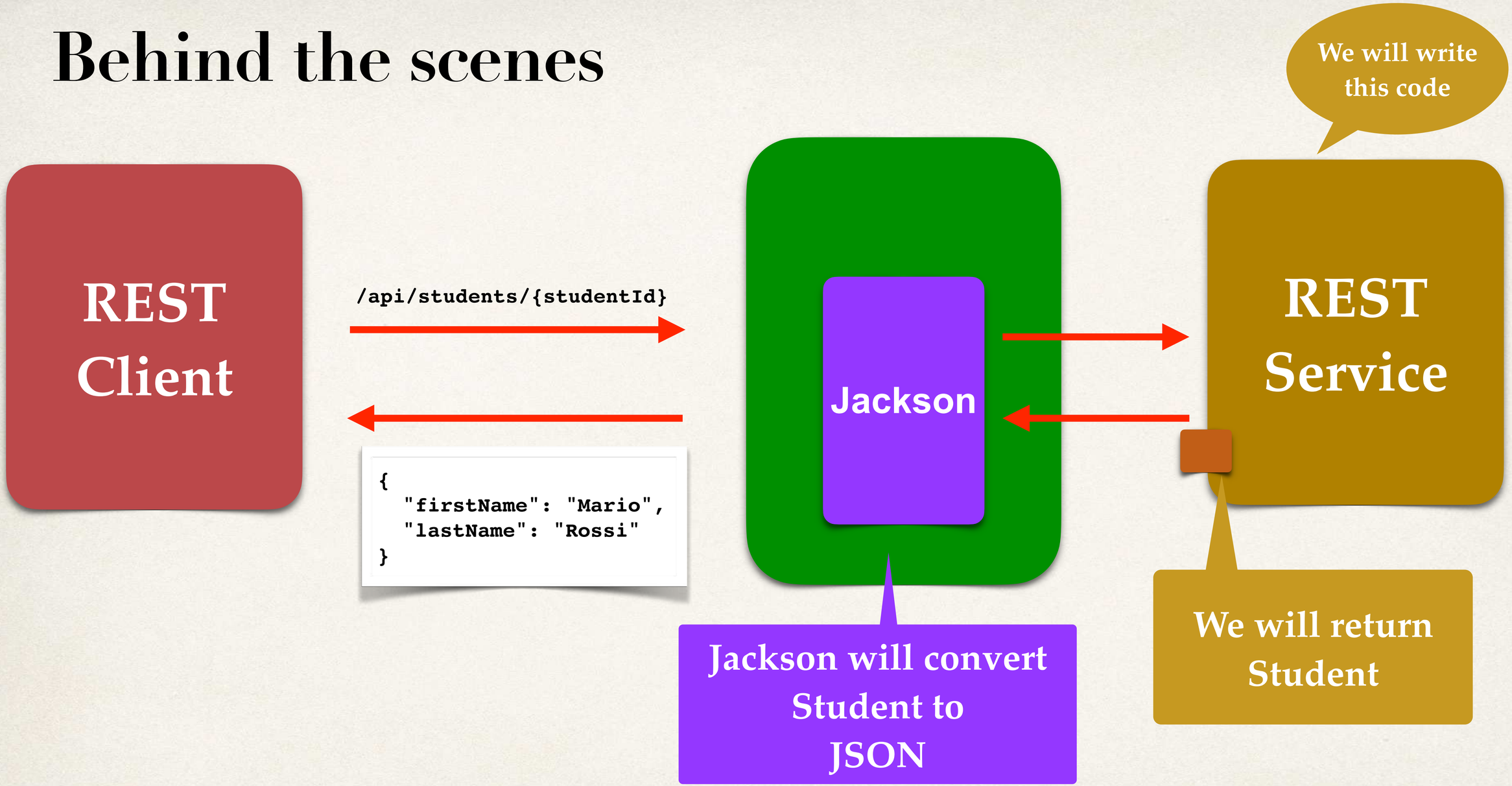
- Retrieve a single student by id



Spring REST Service



Behind the scenes



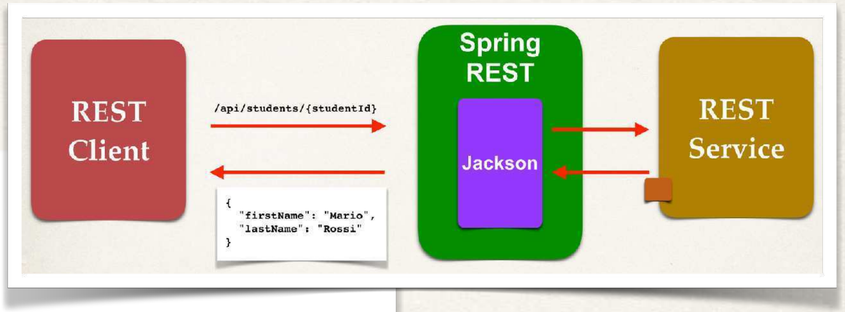
Development Process

Step-By-Step

1. Add request mapping to Spring REST Service

- Bind path variable to method parameter using `@PathVariable`

Step 1: Add Request Mapping



File: StudentRestController.java

```
@RestController
@RequestMapping("/api")
public class StudentRestController {

  // define endpoint for "/students/{studentId}" - return student at index

  @GetMapping("/students/{studentId}")
  public Student getStudent(@PathVariable int studentId) {

    List<Student> theStudents = new ArrayList<>();

    // populate theStudents
    ...


    return theStudents.get(studentId);
  }

}
```

Bind the path variable (by default, must match)

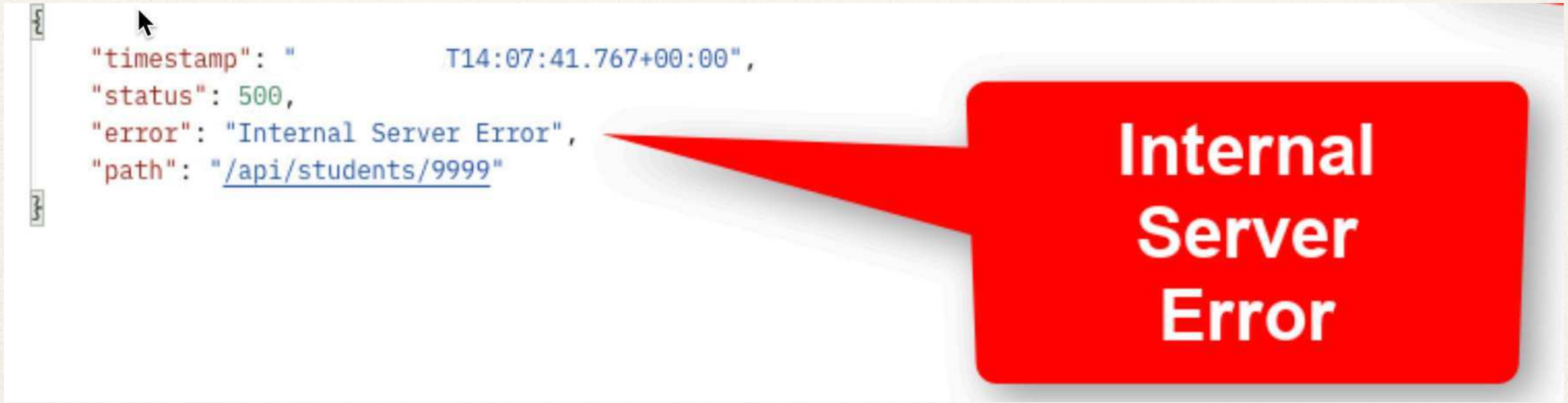
Keep it simple, just index into the list We'll do fancy DB stuff later

Jackson will convert Student to JSON

Spring REST - Exception Handling

Remember our problem?

- Bad student id of 9999 …

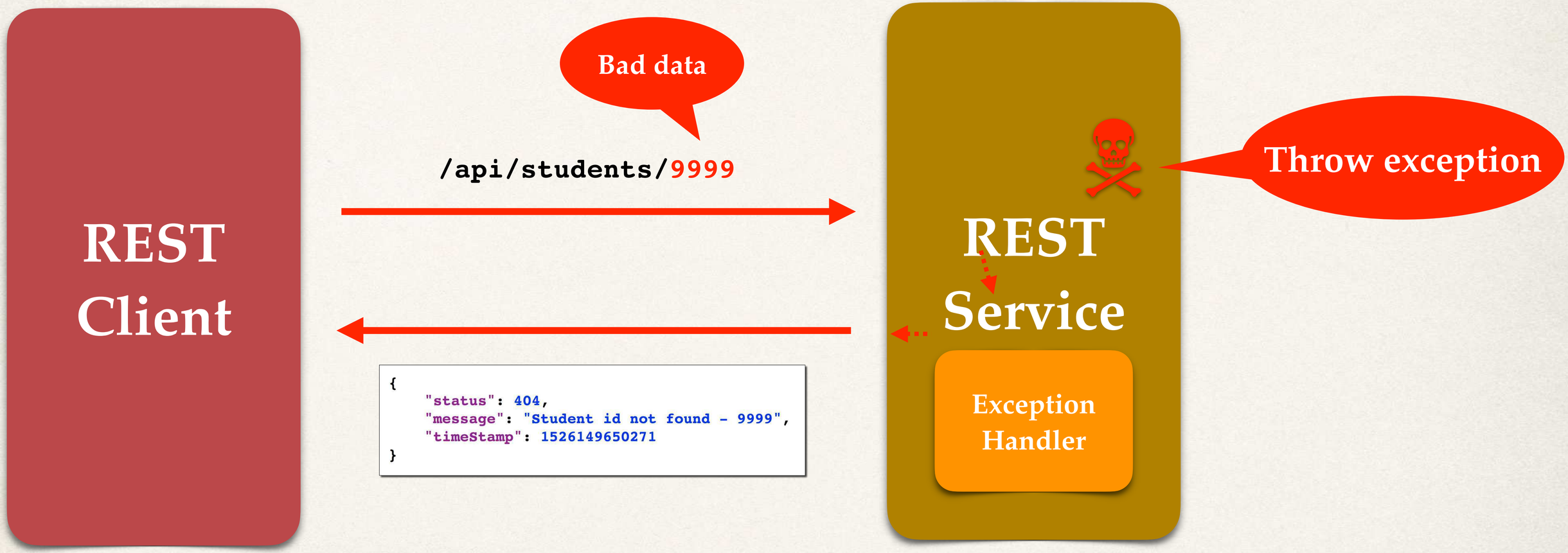


We really want this …

- Handle the exception and return error as JSON

```
{
    "status": 404,
    "message": "Student id not found - 9999",
    "timeStamp": 1526149650271
}
```

Spring REST Exception Handling



Development Process

Step-By-Step

1. Create a custom error response class

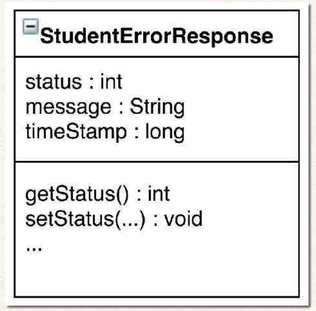
2. Create a custom exception class

3. Update REST service to throw exception if student not found

4. Add an exception handler method using `@ExceptionHandler`

Step 1: Create custom error response class

- The custom error response class will be sent back to client as JSON
- We will define as Java class (POJO)
- Jackson will handle converting it to JSON



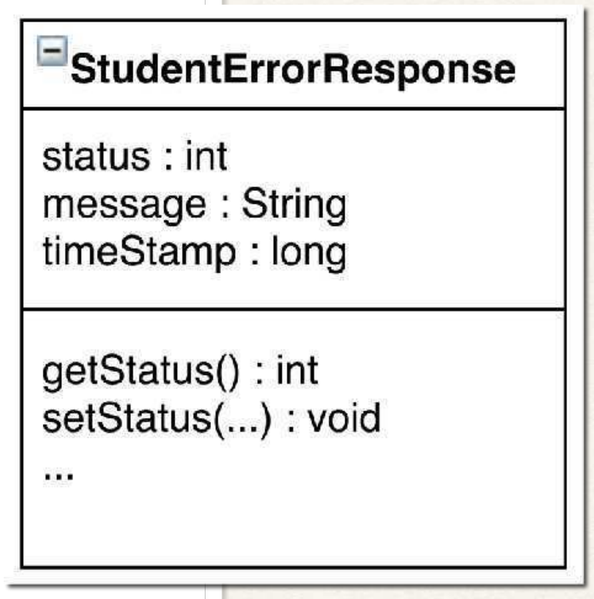
You can define any custom fields that you want to track

```
{
    "status": 404,
    "message": "Student id not found - 9999",
    "timeStamp": 1526149650271
}
```

Step 1: Create custom error response class

File: StudentErrorResponse.java

```
public class StudentErrorResponse {

  private int status;
  private String message;
  private long timeStamp;

  // constructors

  // getters / setters

}
```




```
{
    "status": 404,
    "message": "Student id not found - 9999",
    "timeStamp": 1526149650271
}
```

Step 2: Create custom student exception

- The custom student exception will used by our REST service
- In our code, if we can't find student, then we'll throw an exception
- Need to define a custom student exception class
  - `StudentNotFoundException`

Step 2: Create custom student exception

File: StudentNotFoundException.java

```
public class StudentNotFoundException extends RuntimeException {

  public StudentNotFoundException(String message) {
    super(message);
  }

}
```

Step 3: Update REST service to throw exception

File: StudentRestController.java

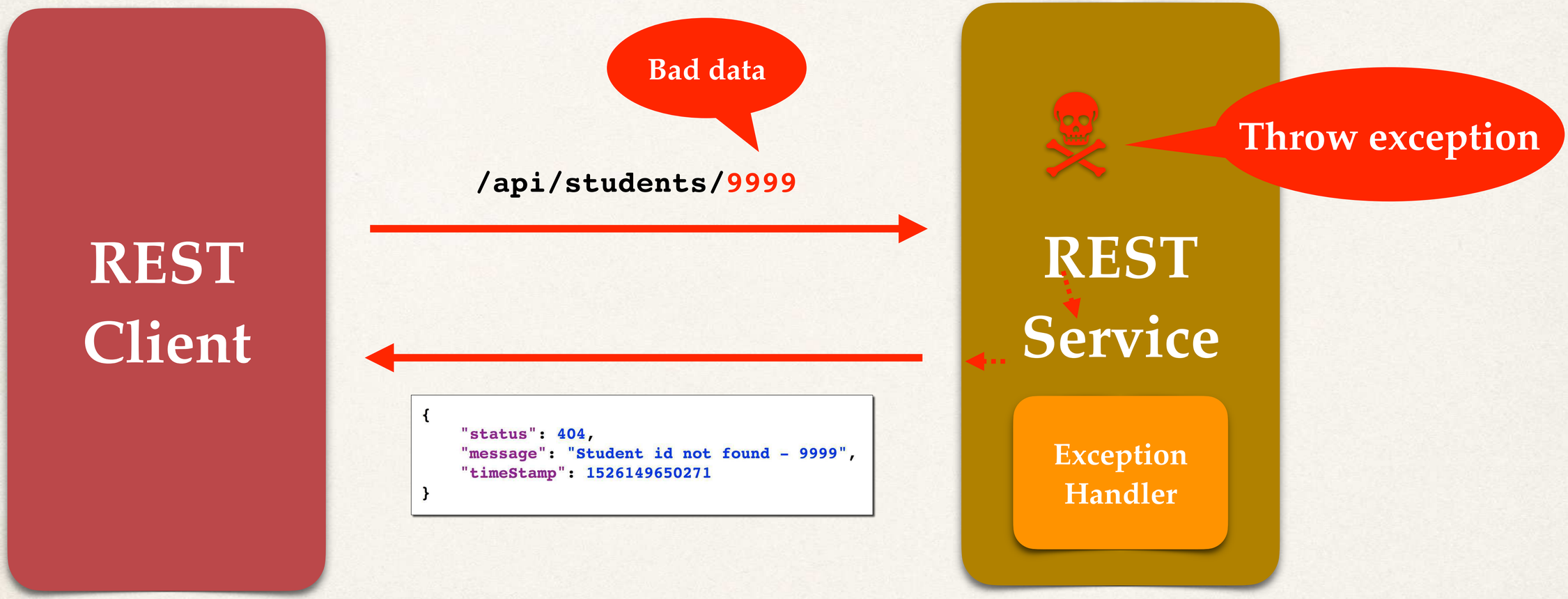
```
@RestController
@RequestMapping("/api")
public class StudentRestController {

   @GetMapping("/students/{studentId}")
   public Student getStudent(@PathVariable int studentId) {

      // check the studentId against list size

      if ( (studentId >= theStudents.size()) || (studentId < 0) ) {
        throw new StudentNotFoundException("Student id not found - " + studentId);
      }

      return theStudents.get(studentId);

   }
   …
}
```

Could also check results from DB

Throw exception

Happy path

Spring REST Exception Handling



Step 4: Add exception handler method

- Define exception handler method(s) with `@ExceptionHandler` annotation
- Exception handler will return a `ResponseEntity`
- `ResponseEntity` is a wrapper for the HTTP response object
- `ResponseEntity` provides fine-grained control to specify:
  - HTTP status code, HTTP headers and Response body

Step 4: Add exception handler method

Exception handler method

Type of the response body

Exception type to handle / catch

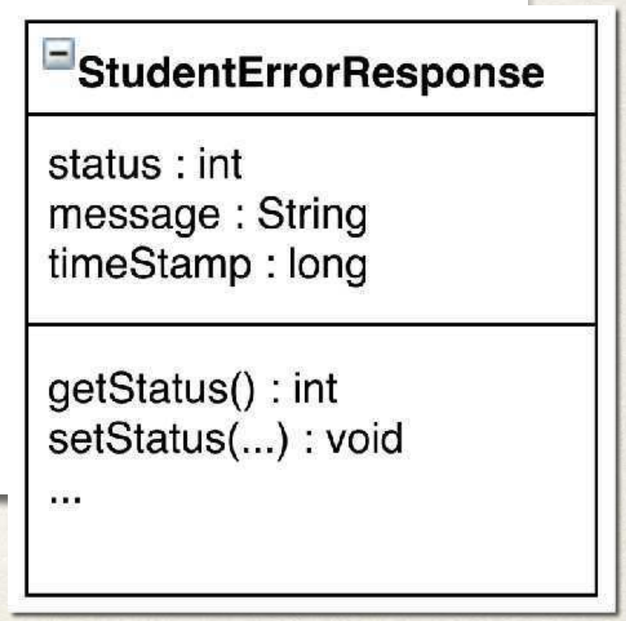
java

```
...pi")
public class StudentRestController {
  ...

  @ExceptionHandler
  public ResponseEntity<StudentErrorResponse> handleException(StudentNotFoundException exc) {

    StudentErrorResponse error = new StudentErrorResponse();

    error.setStatus(HttpStatus.NOT_FOUND.value());
    error.setMessage(exc.getMessage());
    error.setTimeStamp(System.currentTimeMillis());

    return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
  }

}
```

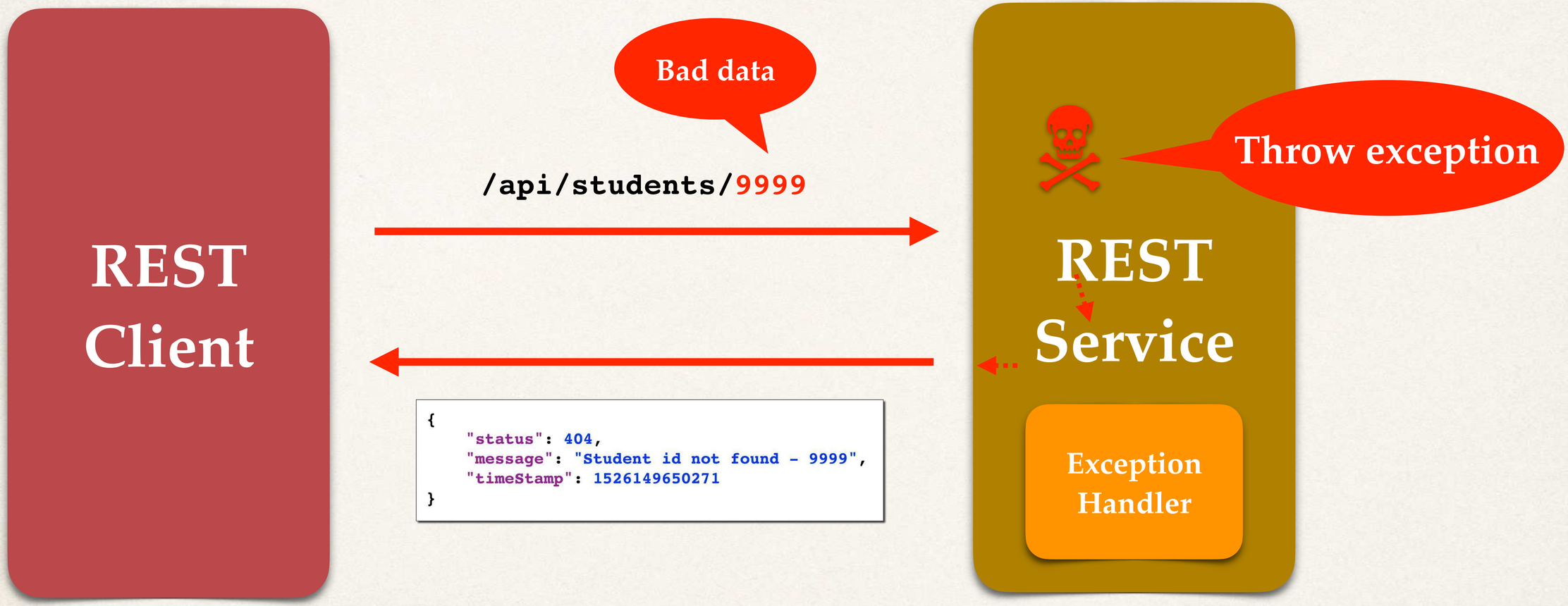
```
{
    "status": 404,
    "message": "Student id not found - 9999",
    "timeStamp": 1526149650271
}
```

Body

Status code

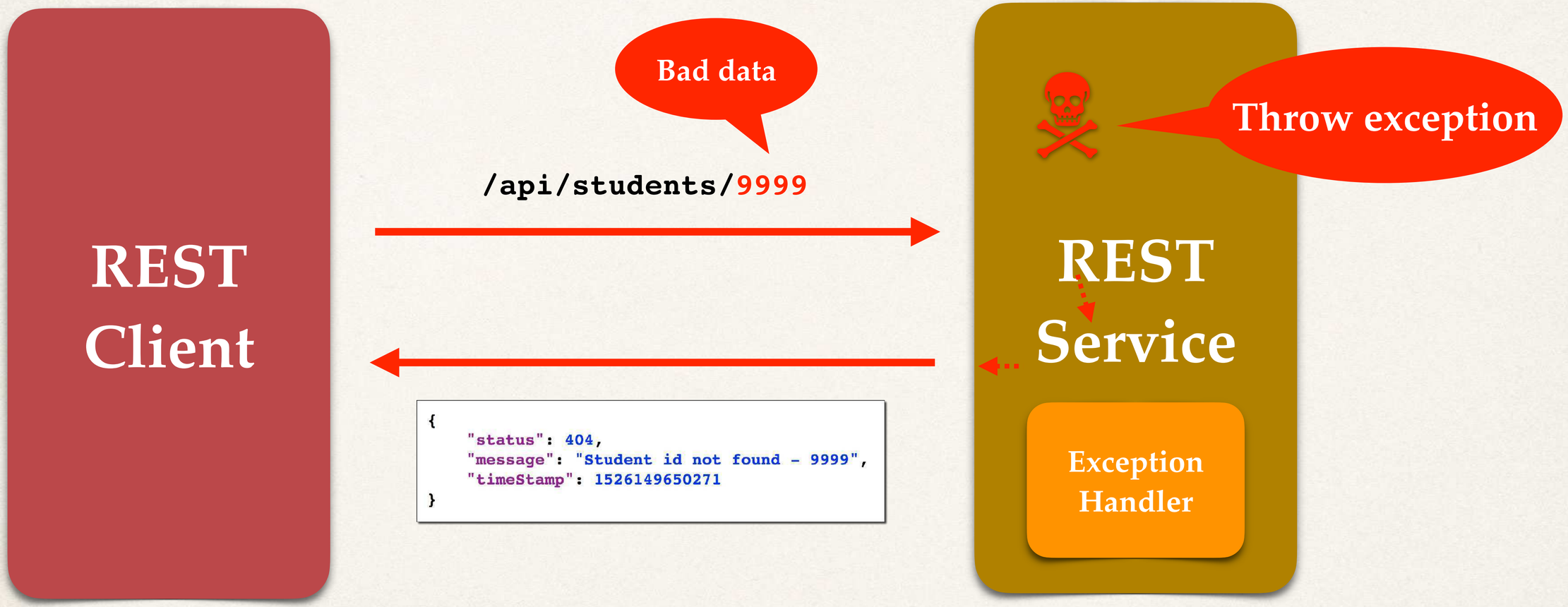


Spring REST Exception Handling



Spring REST - Global Exception Handling

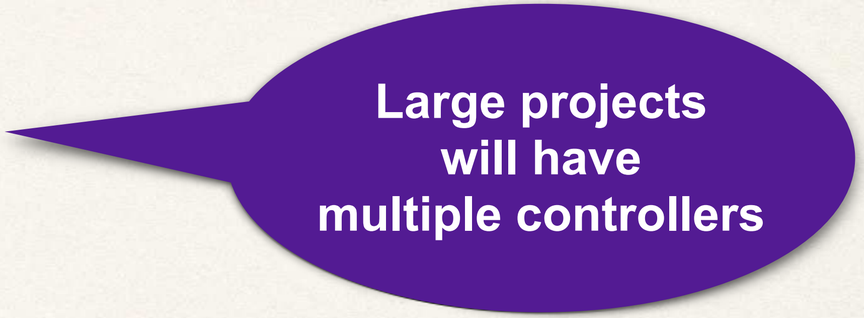
Spring REST Exception Handling



It works, but …

- Exception handler code is only for the specific REST controller

- Can't be reused by other controllers :-(



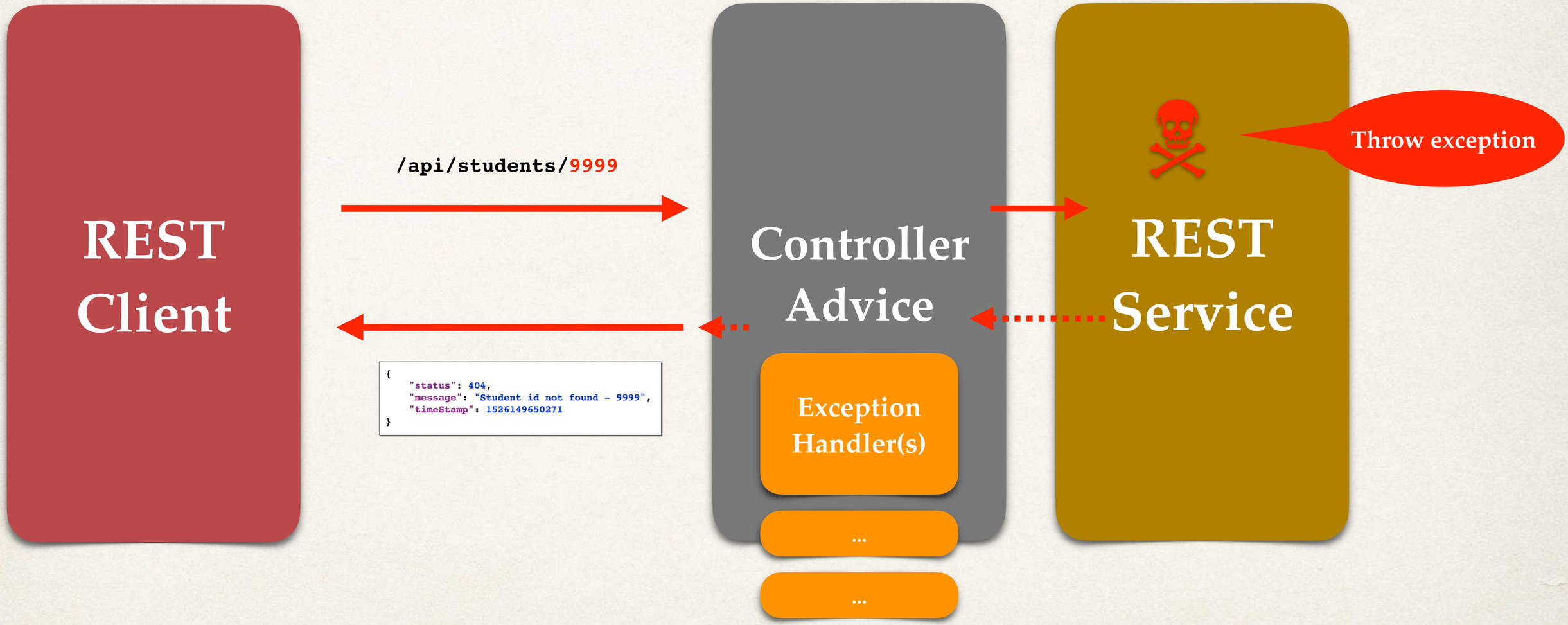
- We need **global** exception handlers
  - Promotes reuse
  - Centralizes exception handling

Spring @ControllerAdvice

- `@ControllerAdvice` is similar to an interceptor / filter
- Pre-process requests to controllers
- Post-process responses to handle exceptions
- Perfect for global exception handling

Real-time use of AOP

Spring REST Exception Handling



Development Process

Step-By-Step

1. Create new @ControllerAdvice

2.Refactor REST service … remove exception handling code

3. Add exception handling code to @ControllerAdvice

Step 1: Create new @ControllerAdvice

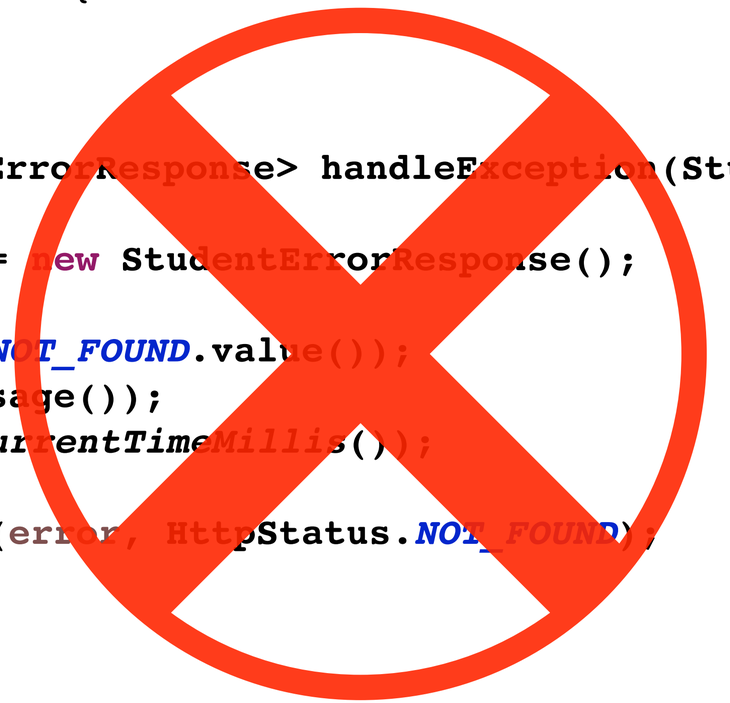
File: StudentRestExceptionHandler.java

```
@ControllerAdvice
public class StudentRestExceptionHandler {

  ...

}
```

Step 2: Refactor - remove exception handling

File: StudentRestController.java

```
@RestController
@RequestMapping("/api")
public class StudentRestController {
   …

   @ExceptionHandler
   public ResponseEntity<StudentErrorResponse> handleException(StudentNotFoundException exc) {

      StudentErrorResponse error = new StudentErrorResponse();

      error.setStatus(HttpStatus.NOT_FOUND.value());
      error.setMessage(exc.getMessage());
      error.setTimeStamp(System.currentTimeMillis());

      return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
   }

}
```



Remove this code

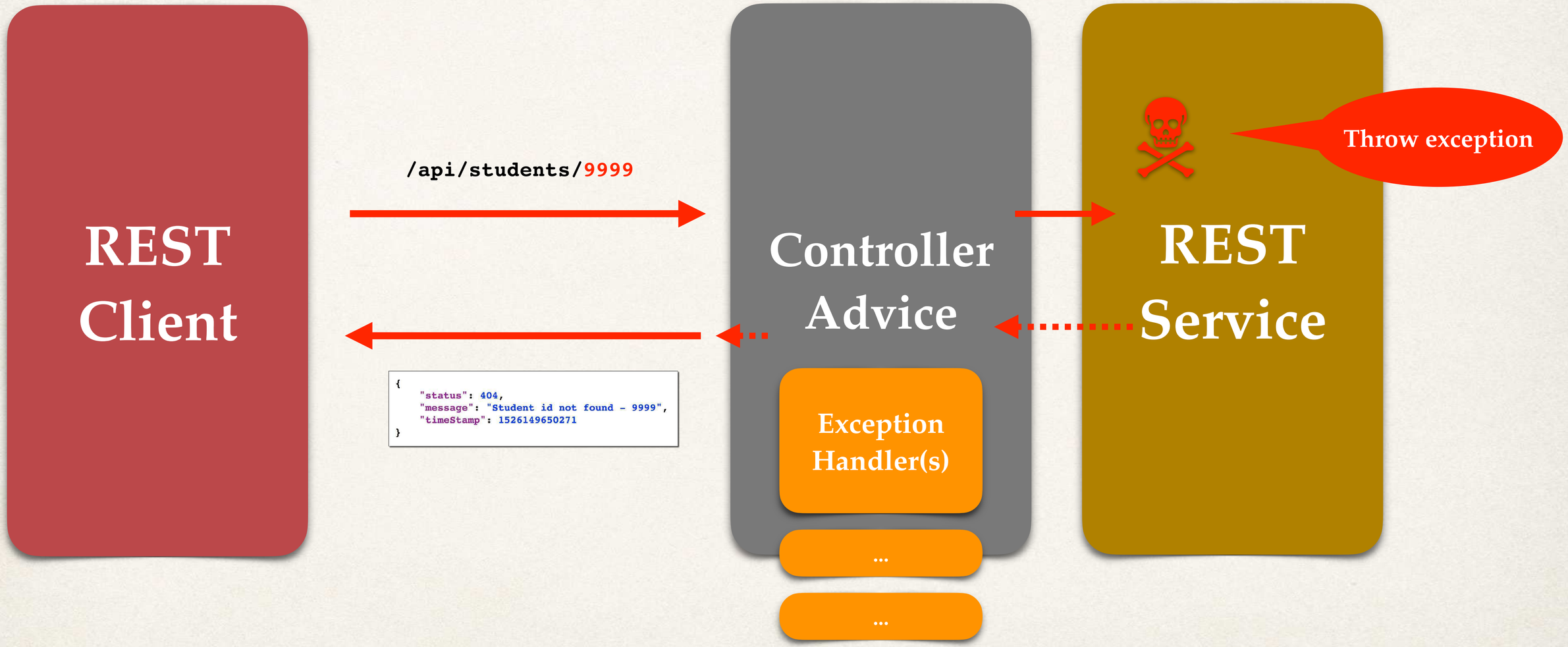
Step 3: Add exception handler to @ControllerAdvice

File: StudentRestExceptionHandler.java

Same code as before

```
@ControllerAdvice
public class StudentRestExceptionHandler {

    @ExceptionHandler
    public ResponseEntity<StudentErrorResponse> handleException(StudentNotFoundException exc) {

        StudentErrorResponse error = new StudentErrorResponse();

        error.setStatus(HttpStatus.NOT_FOUND.value());
        error.setMessage(exc.getMessage());
        error.setTimeStamp(System.currentTimeMillis());

        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
    }

}
```

Spring REST Exception Handling



Spring REST API Design

REST API Design

- For real-time projects, who will use your API?
- Also, how will they use your API?
- Design the API based on requirements

API Design Process

Step-By-Step

1. Review API requirements

2. Identify main resource / entity

3. Use HTTP methods to assign action on resource

API Requirements

From the Boss

Create a REST API for the Employee Directory

REST clients should be able to

- Get a list of employees
- Get a single employee by id
- Add a new employee
- Update an employee
- Delete an employee

Step 2: Identify main resource / entity

- To identify main resource / entity, look for the most prominent "noun"
- For our project, it is "employee"
- Convention is to use plural form of resource / entity: **employees**

```
/api/employees
```

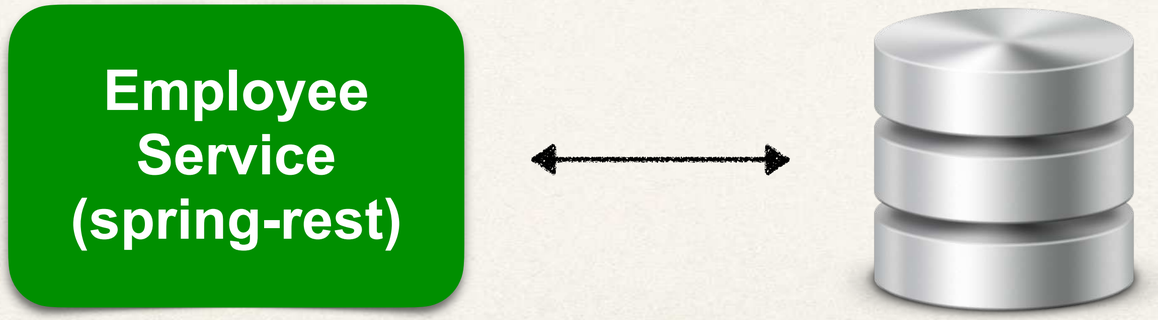
Step 3: Use HTTP methods to assign action on resource

HTTP Method	CRUD Action
POST	Create a new entity
GET	Read a list of entities or single entity
PUT	Update an existing entity
DELETE	Delete an existing entity

Full CRUD

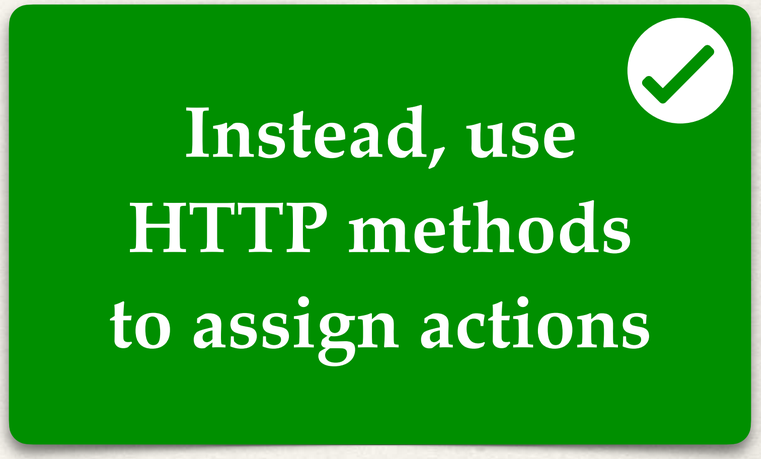
Employee Real-Time Project

HTTP Method	Endpoint	CRUD Action
POST	/api/employees	Create a new employee
GET	/api/employees	Read a list of employees
GET	/api/employees/{employeeId}	Read a single employee
PUT	/api/employees	Update an existing employee
DELETE	/api/employees/{employeeId}	Delete an existing employee



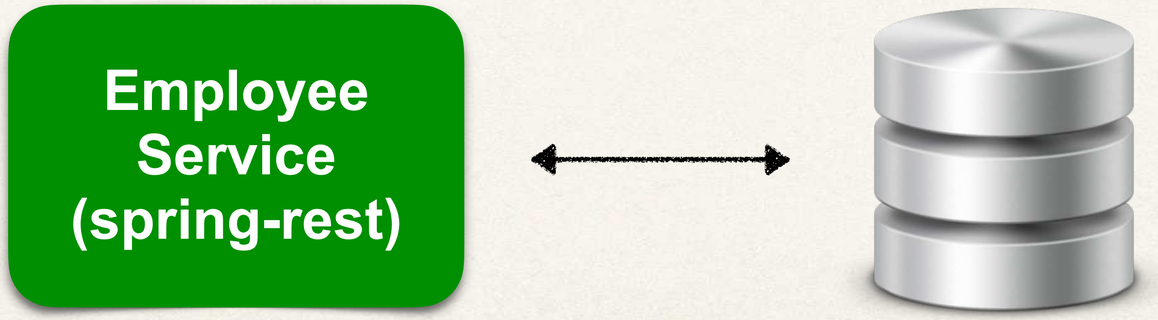
Anti-Patterns

- DO NOT DO THIS … these are REST anti-patterns, bad practice



Employee Real-Time Project

HTTP Method	Endpoint	CRUD Action
POST	/api/employees	Create a new employee
GET	/api/employees	Read a list of employees
GET	/api/employees/{employeeId}	Read a single employee
PUT	/api/employees	Update an existing employee
DELETE	/api/employees/{employeeId}	Delete an existing employee



More API Examples

- On the following slides, we'll look at APIs from other real-time projects

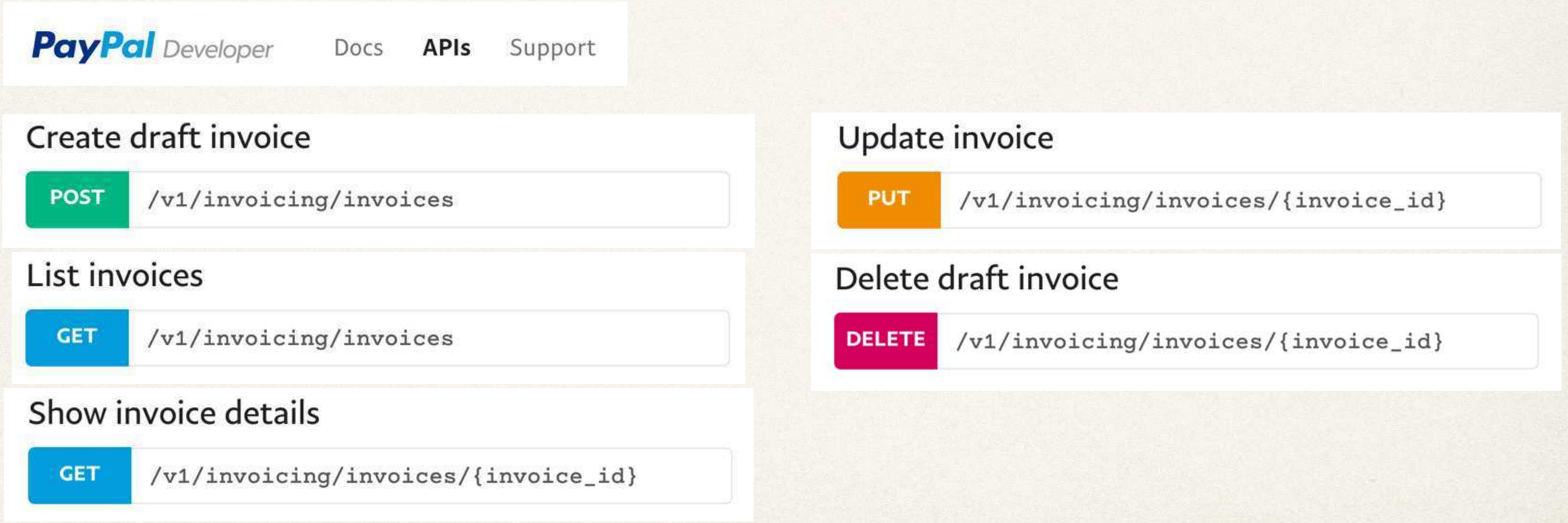
- PayPal

- GitHub

- SalesForce

PayPal

- PayPal Invoicing API
  - `https://developer.paypal.com/docs/api/invoicing/`



GitHub

- GitHub Repositories API
  - `https://developer.github.com/v3/repos/#repositories`

GitHub Developer

Create a new repository

```
POST /user/repos
```

List your repositories

```
GET /user/repos
```

Delete a repository

```
DELETE /repos/:owner/:repo
```

Get a repository

```
GET /repos/:owner/:repo
```

SalesForce REST API

- Industries REST API
  - https://sforce.co/2J40ALH

Retrieve All Individuals

```
GET  /services/apexrest/v1/individual/
```

Retrieve One Individual

```
GET  /services/apexrest/v1/individual/{individual_id}
```

Create an individual

```
POST /services/apexrest/clinic01/v1/individual/
```

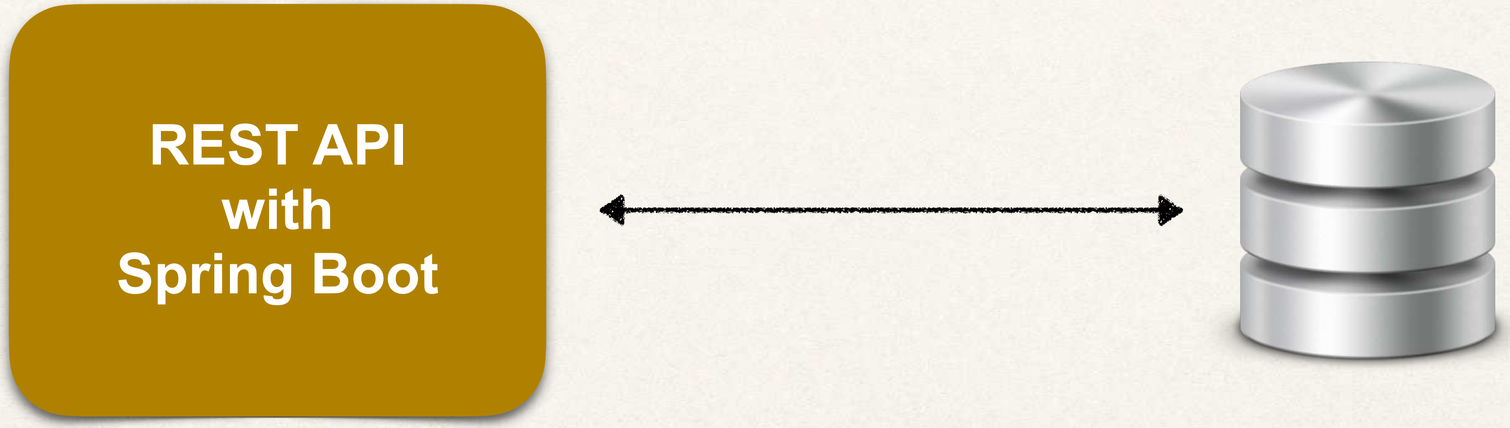
Update an individual

```
PUT  /services/apexrest/clinic01/v1/individual/
```

Spring Boot REST API - Real Time Project

Real-Time Project

REST API with Spring Boot
that connects to a database



API Requirements

From the Boss

Create a REST API for the Employee Directory

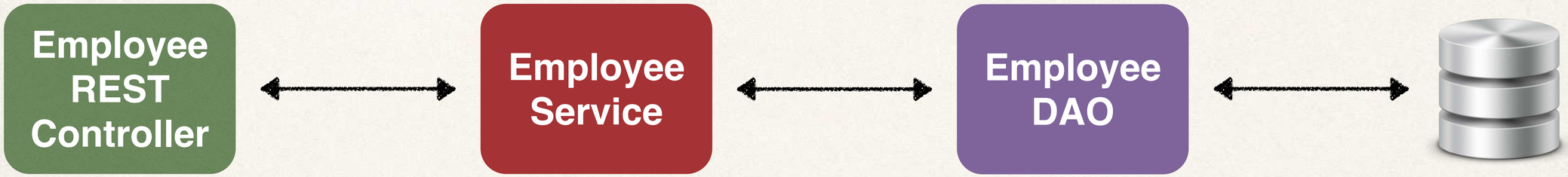
REST clients should be able to

- Get a list of employees
- Get a single employee by id
- Add a new employee
- Update an employee
- Delete an employee

Development Process

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

Application Architecture



Setup Database Table

employee.sql

1. File: employee.sql

2. Create a new database table: **employee**

3. Load table with sample data

Details on downloading the SQL file provided in next section

Creating Spring Boot Project

Development Process

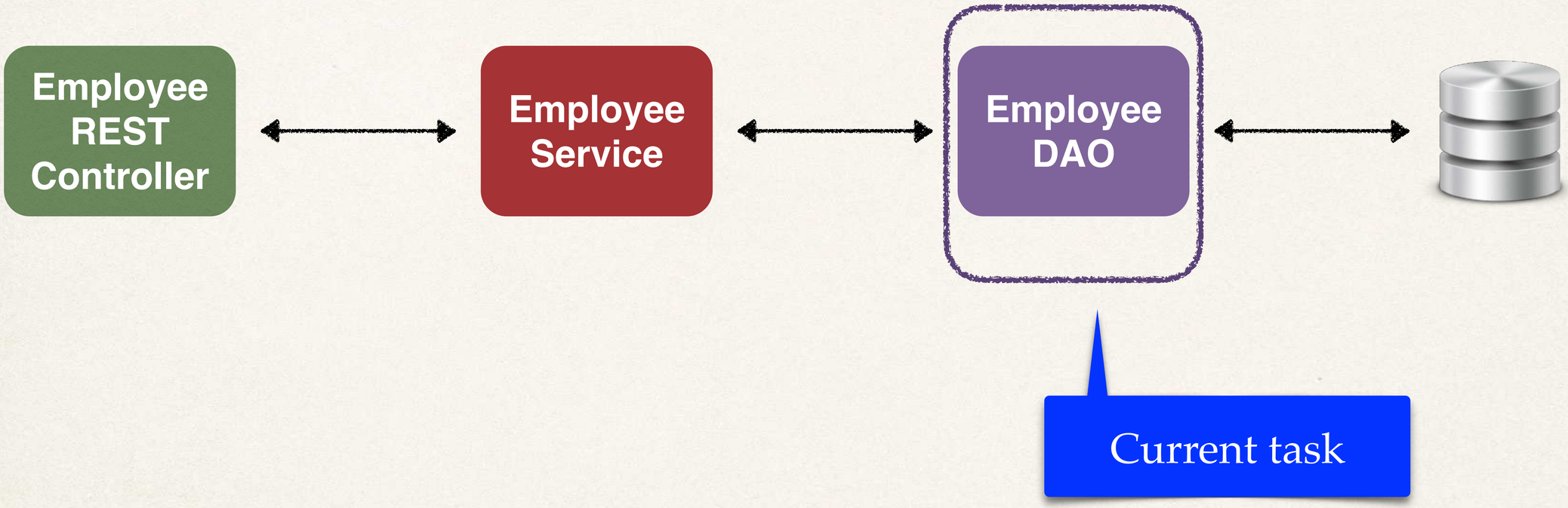
Step-By-Step

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

Our Task

Create DAO in Spring Boot

Application Architecture



Development Process

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

Step-By-Step

Let's build a DAO layer for this

DAO Interface

```
public interface EmployeeDAO {

  List<Employee> findAll();

}
```

DAO Impl

Same interface for consistent API

```
@Repository
public class EmployeeDAOJpaImpl implements EmployeeDAO {

  private EntityManager entityManager;

  @Autowired
  public EmployeeDAOJpaImpl(EntityManager theEntityManager) {
    entityManager = theEntityManager;
  }

 ...
}
```

Automatically created by Spring Boot

Constructor injection

Get a list of employees

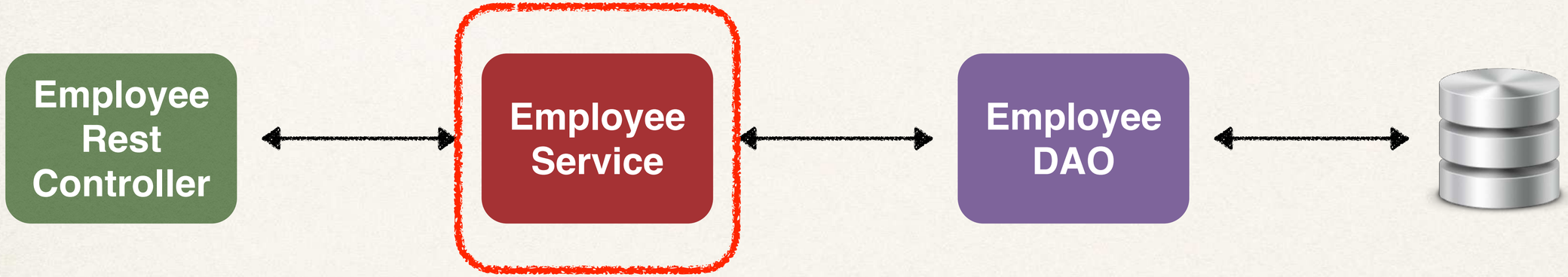
```
@Override
public List<Employee> findAll() {

  // create a query
  TypedQuery<Employee> theQuery =
      entityManager.createQuery("from Employee", Employee.class);

  // execute query and get result list
  List<Employee> employees = theQuery.getResultList();

  // return the results
  return employees;
}
```

Development Process

1. Update db configs in application.properties
2. Create Employee entity
3. Create DAO interface
4. Create DAO implementation
5. Create REST controller to use DAO

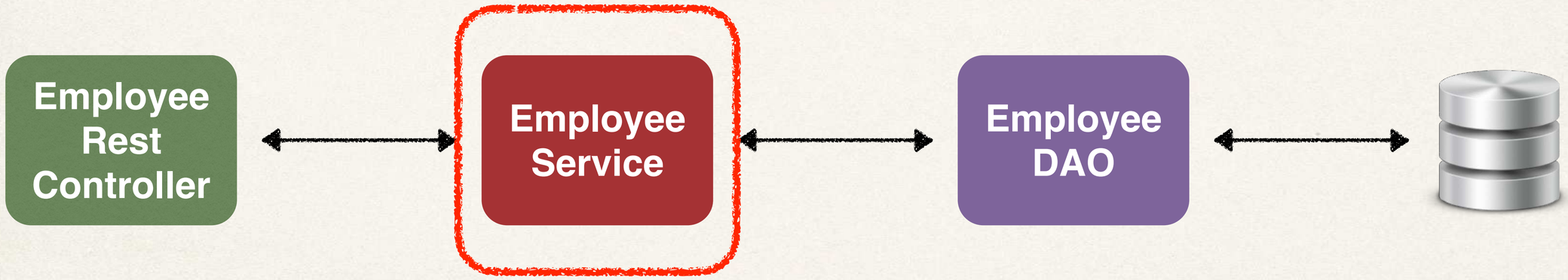
Define Services with @Service

Refactor: Add a Service Layer

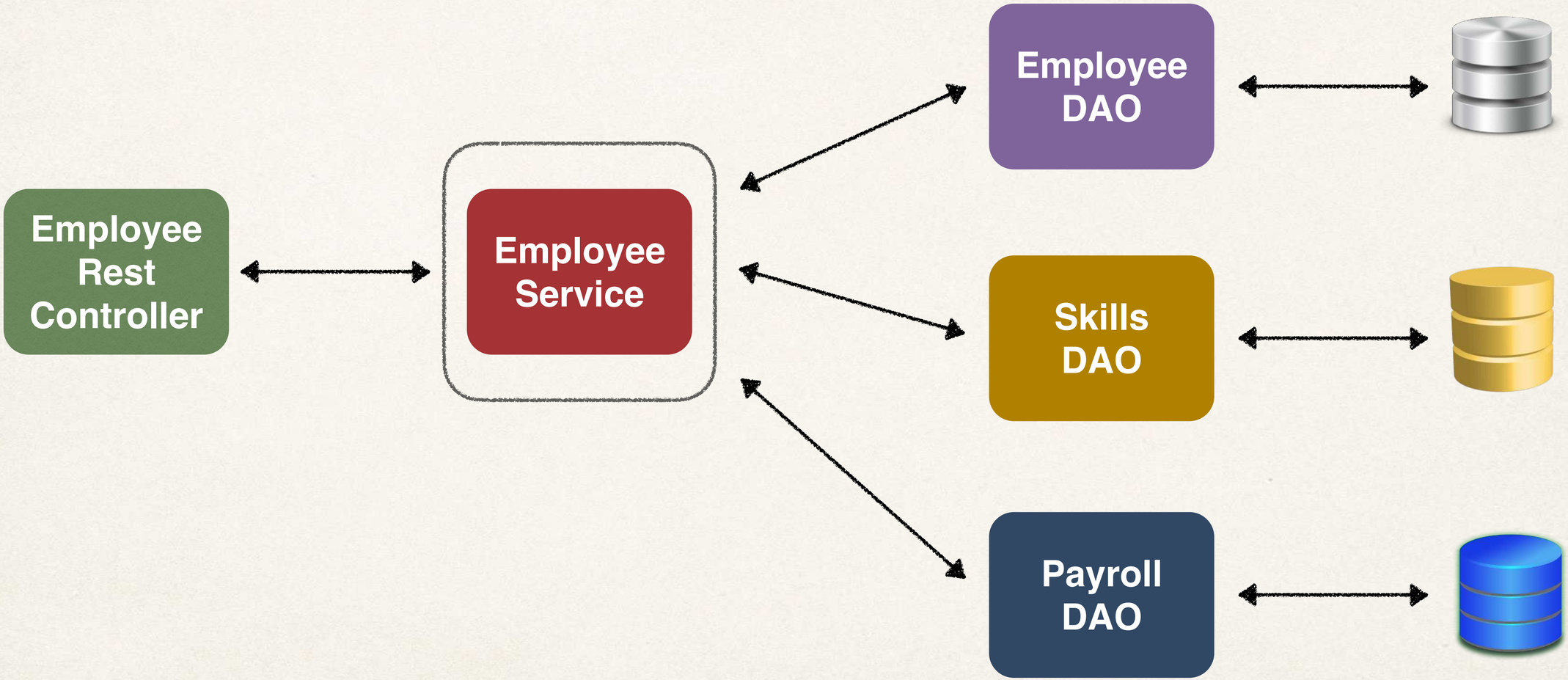


Purpose of Service Layer

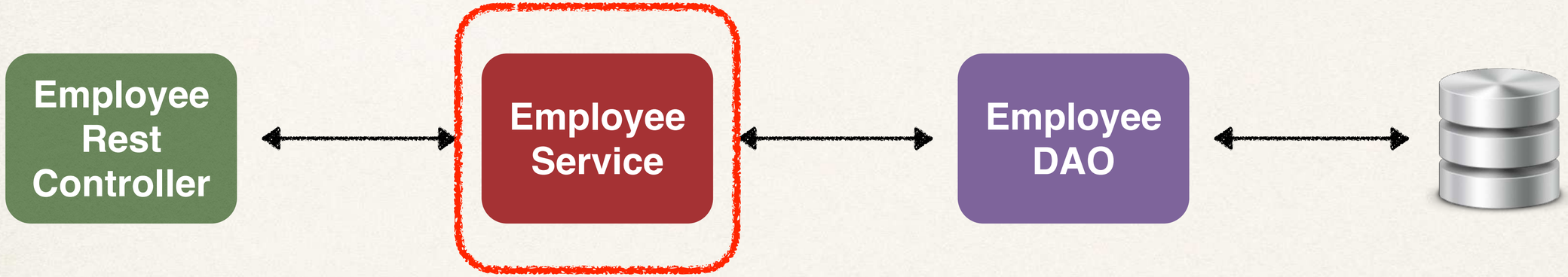
- **Service Facade** design pattern
- Intermediate layer for custom business logic
- Integrate data from multiple sources (DAO/repositories)



Integrate Multiple Data Sources

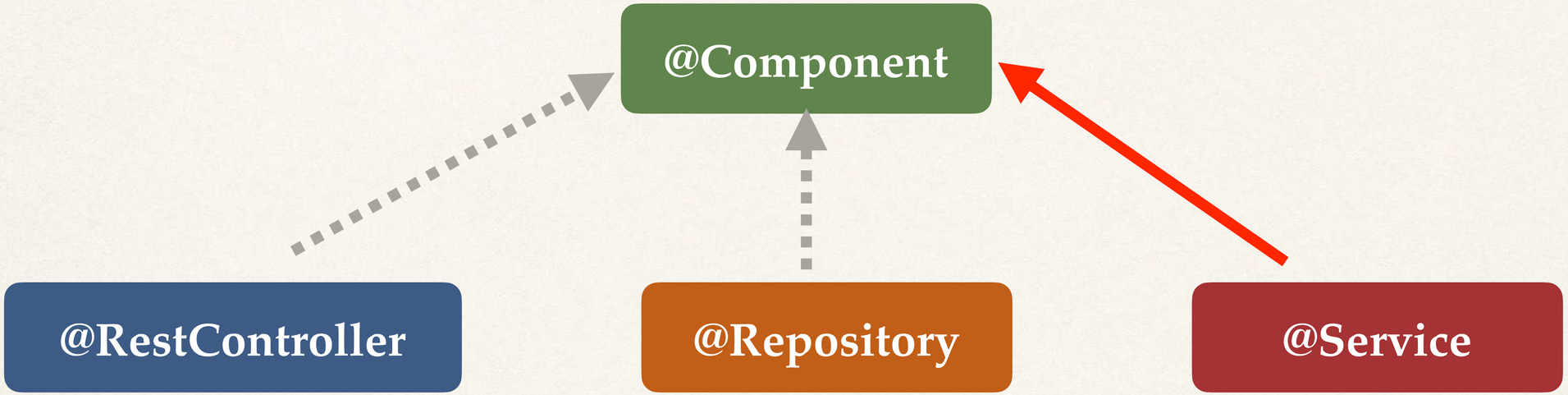


Most Times - Delegate Calls



Specialized Annotation for Services

- Spring provides the **@Service** annotation



Specialized Annotation for Services

- **@Service** applied to Service implementations

- Spring will automatically register the Service implementation
  - thanks to component-scanning

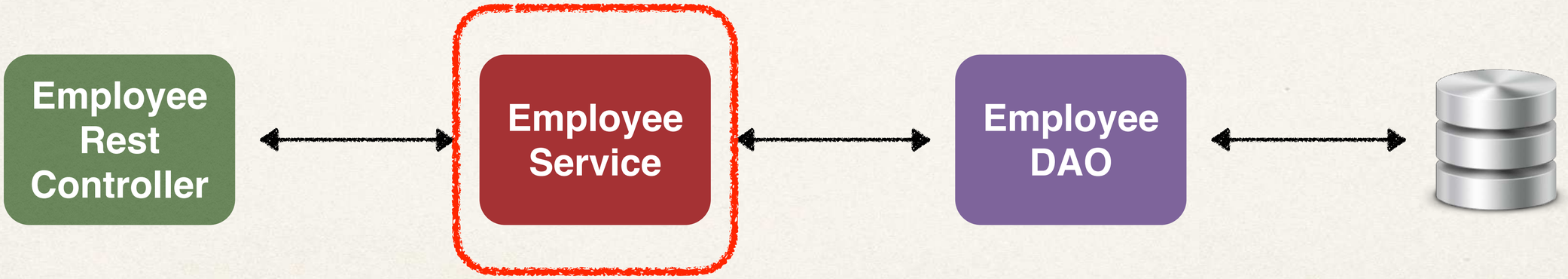
Employee Service

Step-By-Step

1. Define Service interface

2. Define Service implementation

   - Inject the EmployeeDAO



Step 1: Define Service interface

```
public interface EmployeeService {

   List<Employee> findAll();

}
```

Step 2: Define Service implementation

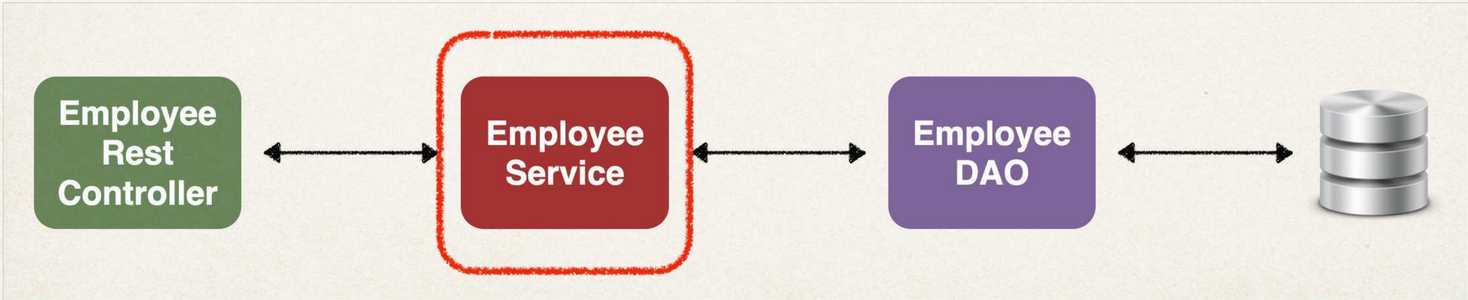
@Service - enables component scanning

```
@Service
public class EmployeeServiceImpl implements EmployeeService {

   // inject EmployeeDAO …

   @Override
   public List<Employee> findAll() {
     return employeeDAO.findAll();
   }
}
```

DAO: Find, Add, Update and Delete

Service Layer - Best Practice

- Best practice is to apply transactional boundaries at the service layer
- It is the service layer's responsibility to manage transaction boundaries
- For implementation code
  - Apply @Transactional on service methods
  - Remove @Transactional on DAO methods if they already exist

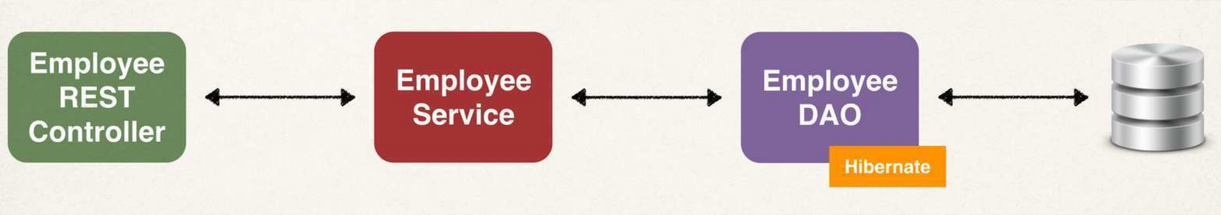


Development Process

Step-By-Step

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

DAO methods



DAO: Get a single employee

```
@Override
public Employee findById(int theId) {

  // get employee
  Employee theEmployee = entityManager.find(Employee.class, theId);

  // return employee
  return theEmployee;
}
```

DAO: Add or Update employee

Note: We don't use @Transactional at DAO layer
It will be handled at Service layer

```
@Override
public Employee save(Employee theEmployee) {

  // save or update the employee
  Employee dbEmployee = entityManager.merge(theEmployee);

  // return dbEmployee
  return dbEmployee;
}
```

if id == 0
then save/insert
else update

Return dbEmployee
It has updated id from the database
(in the case of insert)

DAO: Delete an existing employee

Note: We don't use @Transactional at DAO layer
It will be handled at Service layer

```
@Override
public void deleteById(int theId) {

  // find the employee by id
  Employee theEmployee = entityManager.find(Employee.class, theId);

  // delete the employee
  entityManager.remove(theEmployee);
}
```

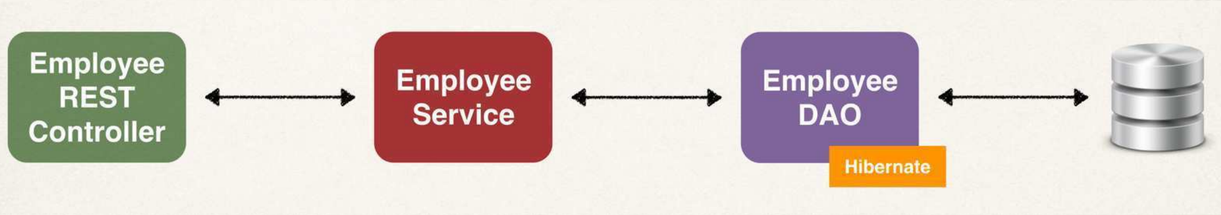
DAO: Find, Add, Update and Delete

Development Process

Step-By-Step

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

DAO methods



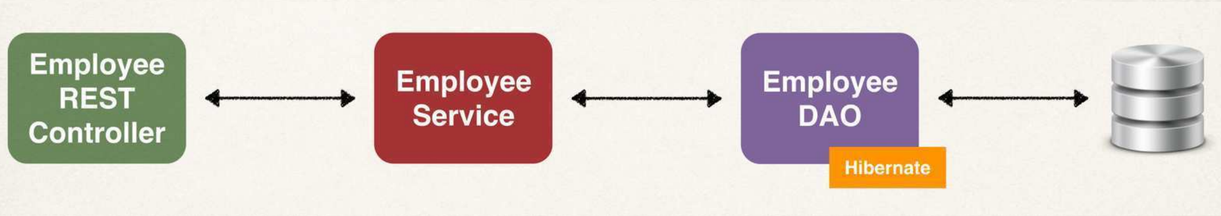
Service: Find, Add, Update and Delete

Development Process

Step-By-Step

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

Service methods



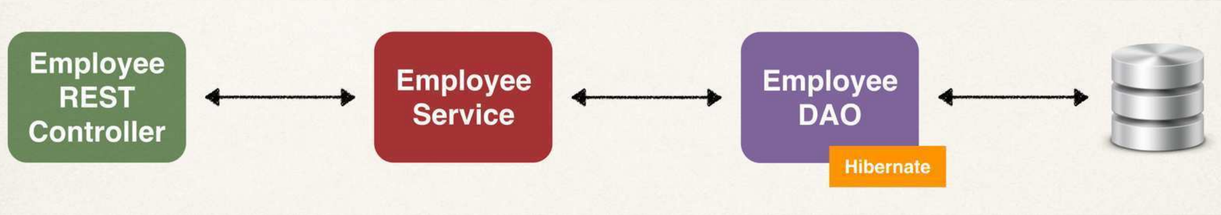
Rest Controller Methods - Find and Add

Development Process

Step-By-Step

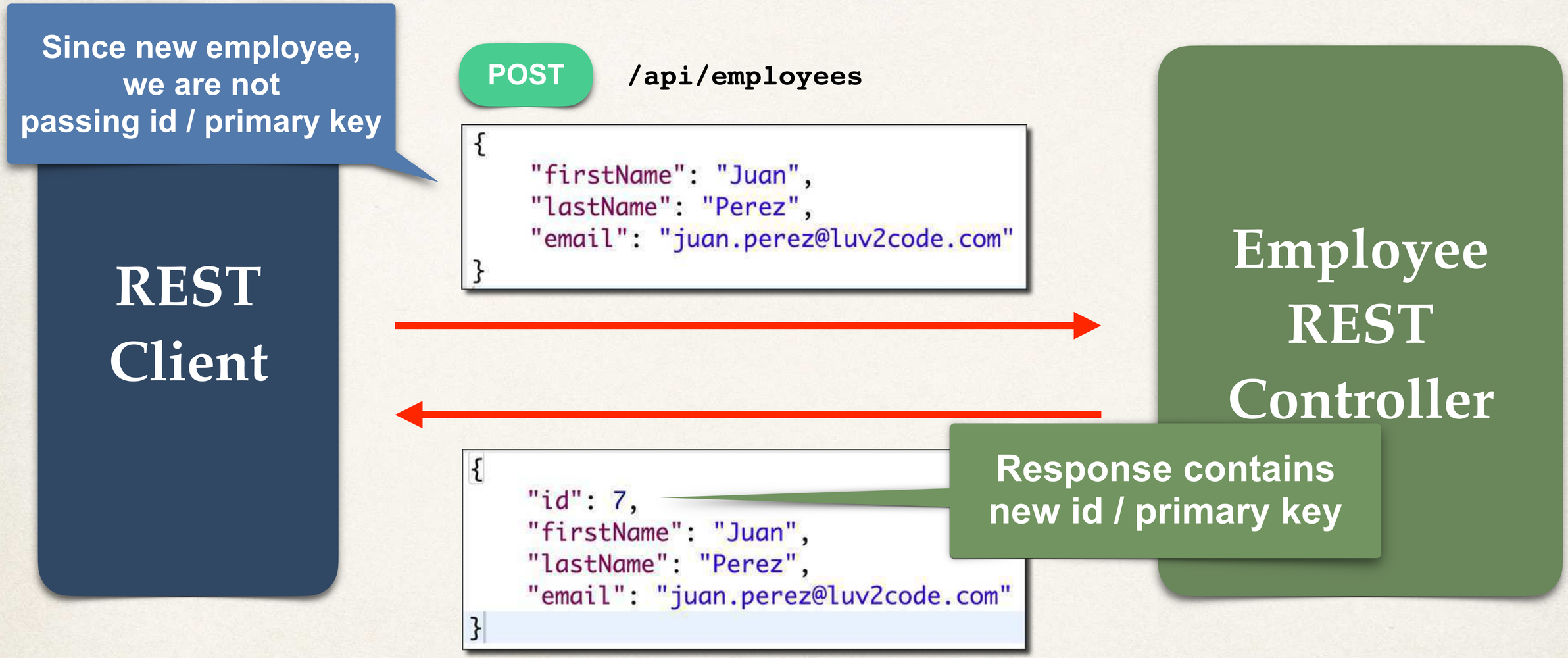
1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

Rest Controller methods



Read a Single Employee

Create a New Employee



Sending JSON to Spring REST Controllers

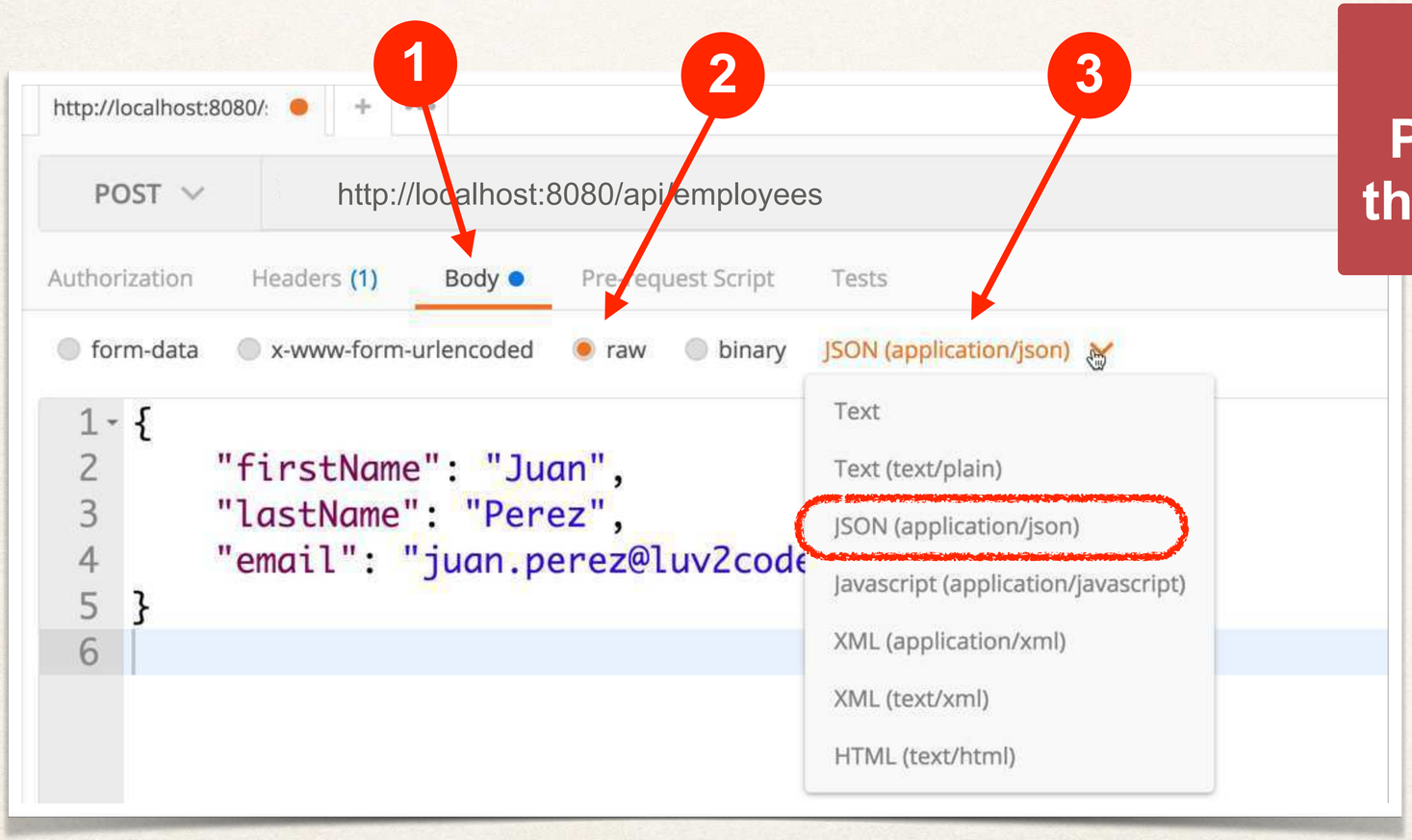
- When sending JSON data to Spring REST Controllers

- For controller to process JSON data, need to set a HTTP request header
  - `Content-type: application/json`

- Need to configure REST client to send the correct HTTP request header

Postman - Sending JSON in Request Body

- Must set HTTP request header in Postman



Based on these configs, Postman will automatically set the correct HTTP request header

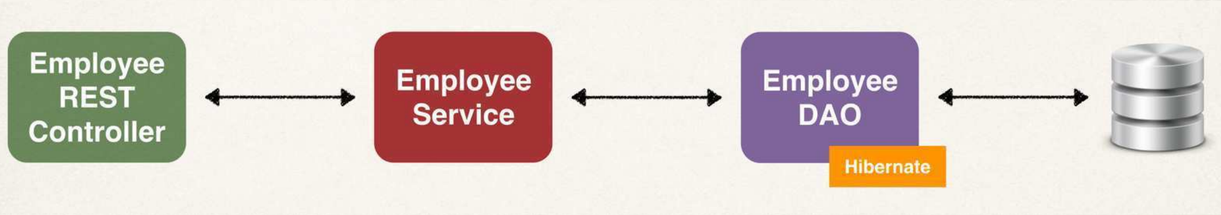
Rest Controller Methods - Update and Delete

Development Process

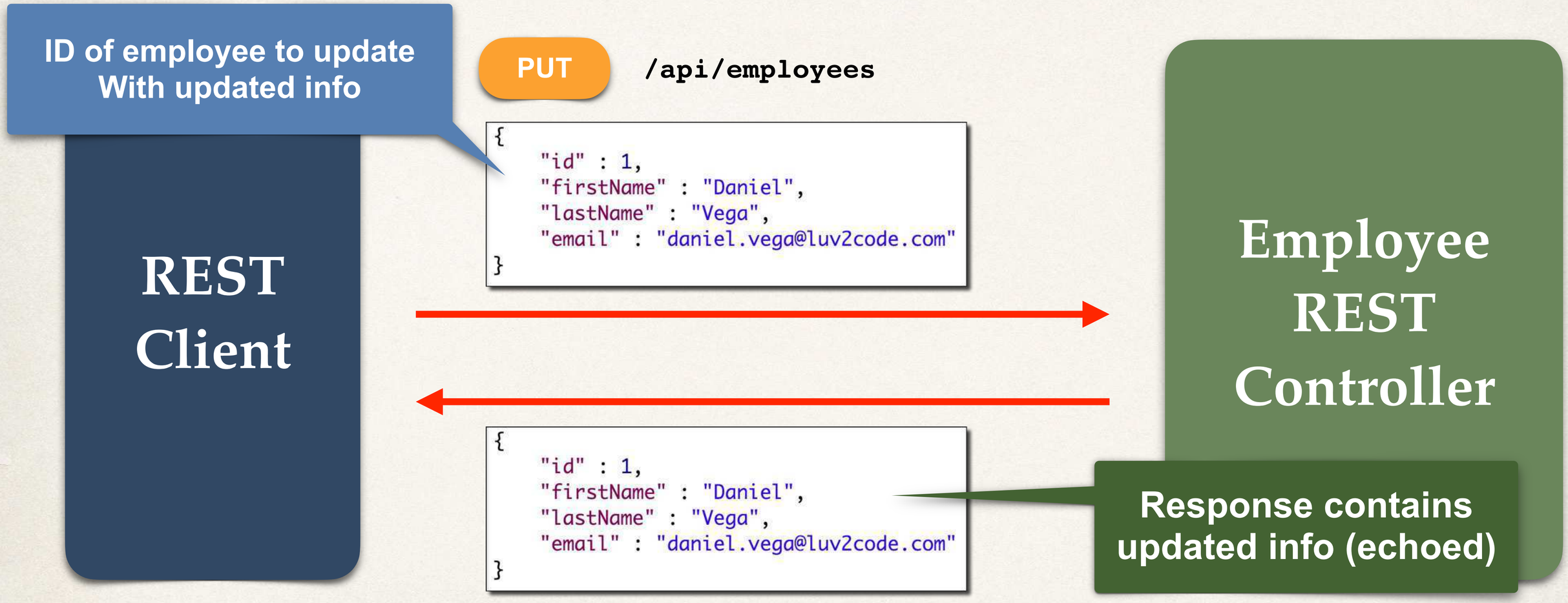
Step-By-Step

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

Rest Controller methods



Update Employee



Delete Employee

Rest Controller Methods
Partial Updates - Patch

Common Pitfall - Partial Update Employee

id	first_name	last_name	email
5	Juan	Vega	juan@luv2code.com

Partial update … only the email address

REST Client

PUT /api/employees

```
{
    "id": 5,
    "email": "vega.juan@demo.com"
}
```

```
{
    "id": 5,
    "firstName": null,
    "lastName": null,
    "email": "vega.juan@demo.com"
}
```

Employee REST Controller

Notice how the other fields are null :-(

Partial Updates - Patch

- For partial updates, need to use **HTTP PATCH**

- Comparison
  - PUT: Replaces the entire resource
  - PATCH: Modifies only specified parts of resource (partial)

- Benefits of **PATCH**
  - Efficiency: Reducing bandwidth by sending only partial changes
  - Flexibility: Allows multiple partial updates in a single request

PATCH - Partial Update Employee

id	first_name	last_name	email
5	Juan	Vega	juan@luv2code.com

Partial update … only the email address

REST Client

PATCH /api/employees/5

REST API convention is to pass the id as a path variable

```
{
    "email": "vega.juan@demo.com"
}
```

```
{
    "id": 5,
    "firstName": "Juan",
    "lastName": "Vega",
    "email": "vega.juan@demo.com"
}
```

Employee REST Controller

Notice how partial updates are applied correctly

PATCH - Partial Update on multiple fields

id	first_name	last_name	email
5	Juan	Vega	juan@luv2code.com

Partial update ...
on multiple fields

REST Client

PATCH /api/employees/5

```
{
    "firstName": "Daniel",
    "email": "vega@demo.com"
}
```

```
{
    "id": 5,
    "firstName": "Daniel", ✓
    "lastName": "Vega",
    "email": "vega@demo.com" ✓
}
```

Employee REST Controller

Notice how partial updates are applied correctly

PATCH - Development Process

1. Inject helper class: ObjectMapper

2. Add support for @PatchMapping request method

3. Apply patch payload to employee

Step 1: Inject helper class: ObjectMapper

- ObjectMapper is a helper class in the Jackson library for JSON processing

- ObjectMapper provides following support
  - Converts Java objects to JSON and vice-versa
  - Allows merging of JSON nodes
  - Provides type safety for conversions: Java <-> JSON

- The ObjectMapper is preconfigured by Spring Boot

Step 1: Inject helper class: ObjectMapper

```
import com.fasterxml.jackson.databind.ObjectMapper;

@RestController
@RequestMapping("/api")
public class EmployeeRestController {

    private ObjectMapper objectMapper;

    @Autowired
    public EmployeeRestController(EmployeeService theEmployeeService, ObjectMapper theObjectMapper) {
        employeeService = theEmployeeService;
        objectMapper = theObjectMapper;
    }
    …

}
```

Helper class from Jackson library

ObjectMapper is auto-configured by Spring Boot for JSON processing

Step 2: Add support for @PatchMapping request method

```
@PatchMapping("/employees/{employeeId}")
public Employee patchEmployee(@PathVariable int employeeId, @RequestBody Map<String, Object> patchPayload) {

    Employee tempEmployee = employeeService.findById(employeeId);

    // throw exception if null
    if (tempEmployee == null) {
        throw new RuntimeException("Employee id not found - " + employeeId);
    }

    // throw exception if request body updates contains id
    if (patchPayload.containsKey("id")) {
        throw new RuntimeException("Employee id not allowed in request body - " + employeeId);
    }

    Employee patchedEmployee = apply(patchPayload, tempEmployee);

    Employee dbEmployee = employeeService.save(patchedEmployee);

    return dbEmployee;
}
```

The “id” field is not allowed in the payload
We will not change primary key

Apply the partial patch updates

Method defined in next step …

Step 3: Apply patch payload to employee

```
private Employee apply(Map<String, Object> patchPayload, Employee tempEmployee) {

    // Convert employee object to a JSON object node
    ObjectNode employeeNode = objectMapper.convertValue(tempEmployee, ObjectNode.class);

    // Convert the patchPayload map to a JSON object node
    ObjectNode patchNode = objectMapper.convertValue(patchPayload, ObjectNode.class);

    // Merge the patch updates into the employee node
    employeeNode.setAll(patchNode);

    return objectMapper.convertValue(employeeNode, Employee.class);
}
```

Convert JSON object node back to Employee object

PATCH - Partial Update Employee

id	first_name	last_name	email
5	Juan	Vega	juan@luv2code.com

Partial update … only the email address

REST Client

PATCH /api/employees/5

```
{
    "email": "vega.juan@demo.com"
}
```

```
{
    "id": 5,
    "firstName": "Juan",
    "lastName": "Vega",
    "email": "vega.juan@demo.com"
}
```

Employee REST Controller

Notice how partial updates are applied correctly

PATCH

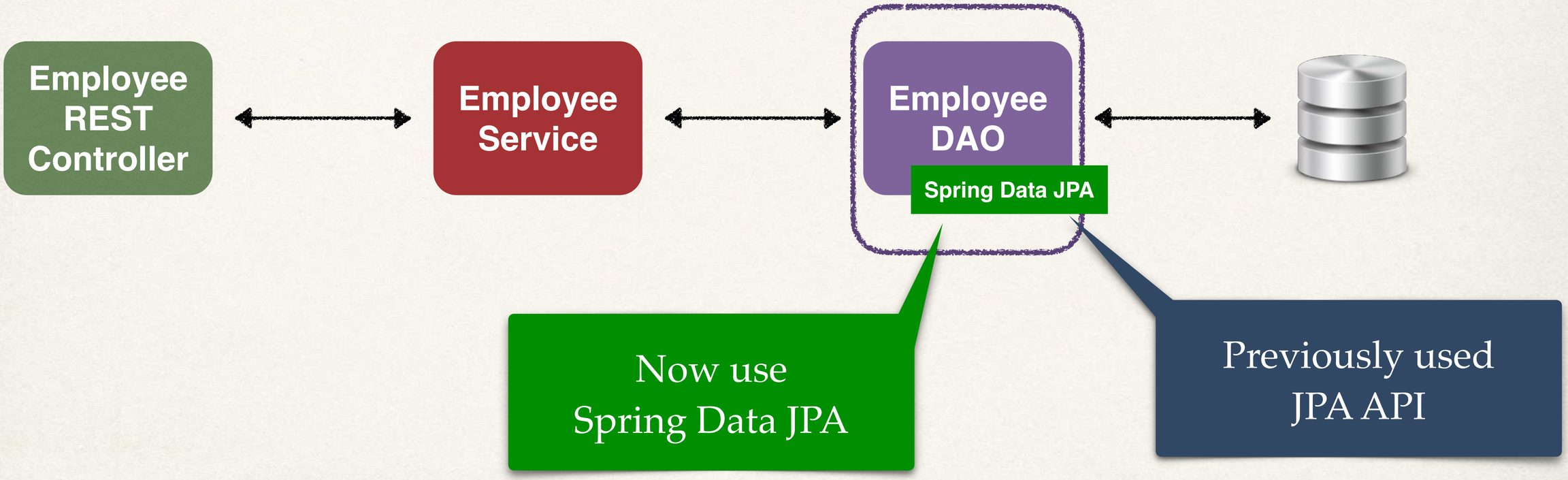
- The approach shown in the previous slides covers the majority of use cases for partial updates

- However, if you have complex use cases
  - Deeply nested JSON entities
  - Add, move, remove, copy fields
  - Move / manipulate array elements
  - Complex transformations / data enrichment

- RFC 6902 - JSON Patch - `https://www.rfc-editor.org/rfc/rfc6902.html`
- RFC 7386 - JSON Merge Patch - `https://www.rfc-editor.org/rfc/rfc7386.html`
- json-patch project - `https://github.com/java-json-tools/json-patch`

Spring Data JPA in Spring Boot

Application Architecture



The Problem

- We saw how to create a DAO for **`Employee`**

```
public interface EmployeeDAO {

    public List<Employee> findAll();

    public Employee findById(int theId);

    public void save(Employee theEmployee);

    public void deleteById(int theId);

}
```

- What if we need to create a DAO for another entity?
  - **`Customer`**, **`Student`**, **`Product`**, **`Book`** ...

```
@Repository
public class EmployeeDAOJpaImpl implements EmployeeDAO {

    private EntityManager entityManager;

    @Autowired
    public EmployeeDAOJpaImpl(EntityManager theEntityManager) {
        entityManager = theEntityManager;
    }

    @Override
    public List<Employee> findAll() {

        // create a query
        TypedQuery<Employee> theQuery =
                entityManager.createQuery("from Employee", Employee.class);

        // execute query and get result list
        List<Employee> employees = theQuery.getResultList();

        // return the results
        return employees;
    }

    @Override
    public Employee findById(int theId) {

        // get employee
        Employee theEmployee =
                entityManager.find(Employee.class, theId);

        // return employee
        return theEmployee;
    }
```

- Do we have to repeat all of the same code again???

Creating DAO

- You may have noticed a pattern with creating DAOs

```
@Override
public Employee findById(int theId) {

  // get data
  Employee theData = entityManager.find(Employee.class, theId);

  // return data
  return theData;
}
```

Most of the code is the same

Only difference is the entity type and primary key

Entity type

Primary key

My Wish

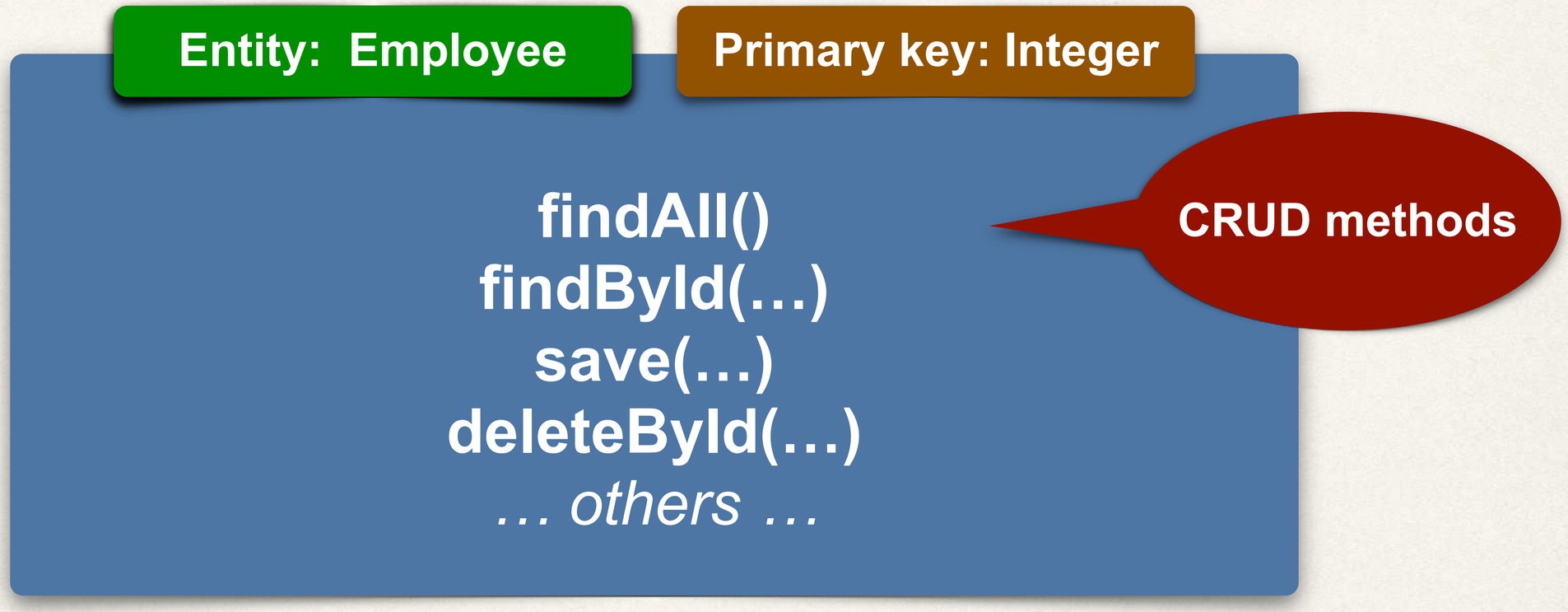
- I wish we could tell Spring:

Create a DAO for me

Plug in my entity type and primary key

Give me all of the basic CRUD features for free

My Wish Diagram



Spring Data JPA - Solution

- **Spring Data JPA** is the solution!!!!

https://spring.io/projects/spring-data-jpa

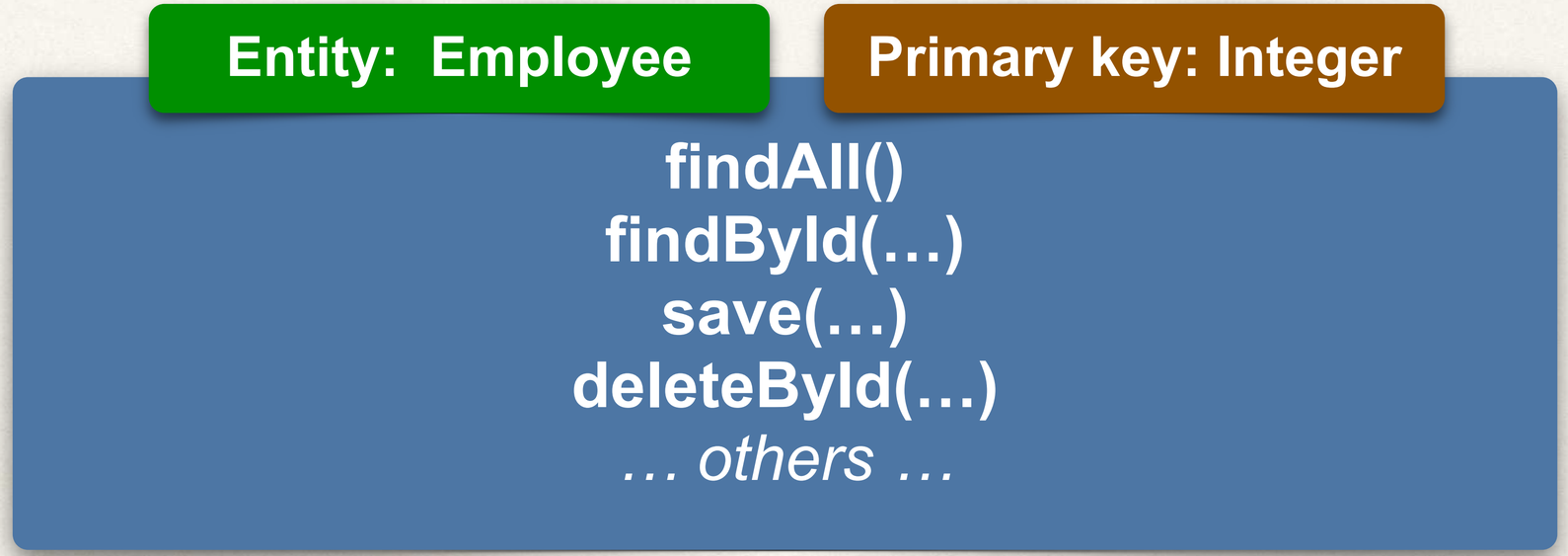
- Create a DAO and just plug in your entity type and primary key

- Spring will give you a CRUD implementation for FREE …. like MAGIC!!
  - Helps to minimize boiler-plate DAO code … yaaay!!!

More than 70% reduction in code … depending on use case

JpaRepository

- Spring Data JPA provides the interface: **`JpaRepository`**
- Exposes methods (some by inheritance from parents)



Development Process

Step-By-Step

1. Extend **`JpaRepository`** interface

2. Use your Repository in your app

No need for implementation class

Step 1: Extend JpaRepository interface

Entity type

Primary key

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

  // that's it ... no need to write any code LOL!

}
```

No need for implementation class

Get these methods for free

Entity: Employee

Primary key: Integer

findAll()
findById(…)
save(…)
deleteById(…)
… others …

JpaRepository Docs

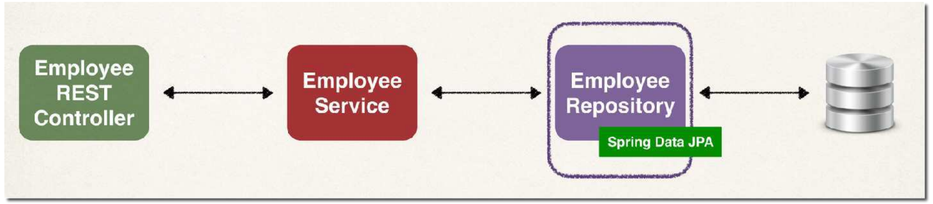
- Full list of methods available … see JavaDoc for **`JpaRepository`**

www.luv2code.com/jpa-repository-javadoc

Step 2: Use Repository in your app

```
@Service
public class EmployeeServiceImpl implements EmployeeService {

  private EmployeeRepository employeeRepository;

  @Autowired
  public EmployeeServiceImpl(EmployeeRepository theEmployeeRepository) {
    employeeRepository = theEmployeeRepository;
  }

  @Override
  public List<Employee> findAll() {
    return employeeRepository.findAll();
  }
  ...
}
```

Our repository



Magic method that is available via repository

Minimized Boilerplate Code

Before Spring Data JPA

```
public interface EmployeeDAO {

    public List<Employee> findAll();

    public Emplo      yId(int theId);

    public void       ee theEmployee

    public void d     theId)

}
```

```
@Repository
public class Employee

    private EntityManag

    @Autowired
    public EmployeeDAOJpa      tyManager) {
        entityManager = th
    }

    @Override
    public List<Empl

        // create
        TypedQue                    loyee.class);

        //
        L                  = theQ

    }

    @Ov
    publ          dById(int theId) {

        // get
        Employe       eEmployee =
                entityManager.find(Employee.class, theId);

        // return employee
        return theEmployee;
    }
```

2 Files
30+ lines of code

After Spring Data JPA

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

    // that's it ... no need to write any code LOL!

}
```

1 File
3 lines of code!

No need for implementation class

Advanced Features

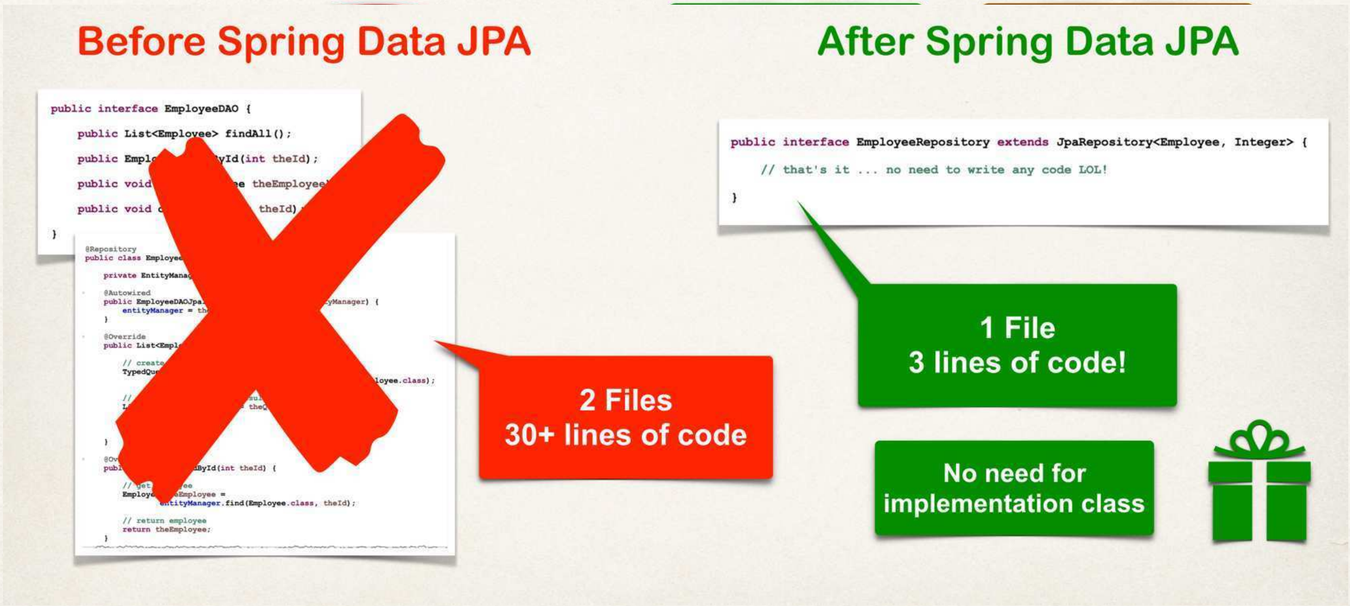
- Advanced features available for
  - Extending and adding custom queries with JPQL
  - Query Domain Specific Language (Query DSL)
  - Defining custom methods (low-level coding)

www.luv2code.com/spring-data-jpa-defining-custom-queries

Spring Data REST in Spring Boot

Spring Data JPA

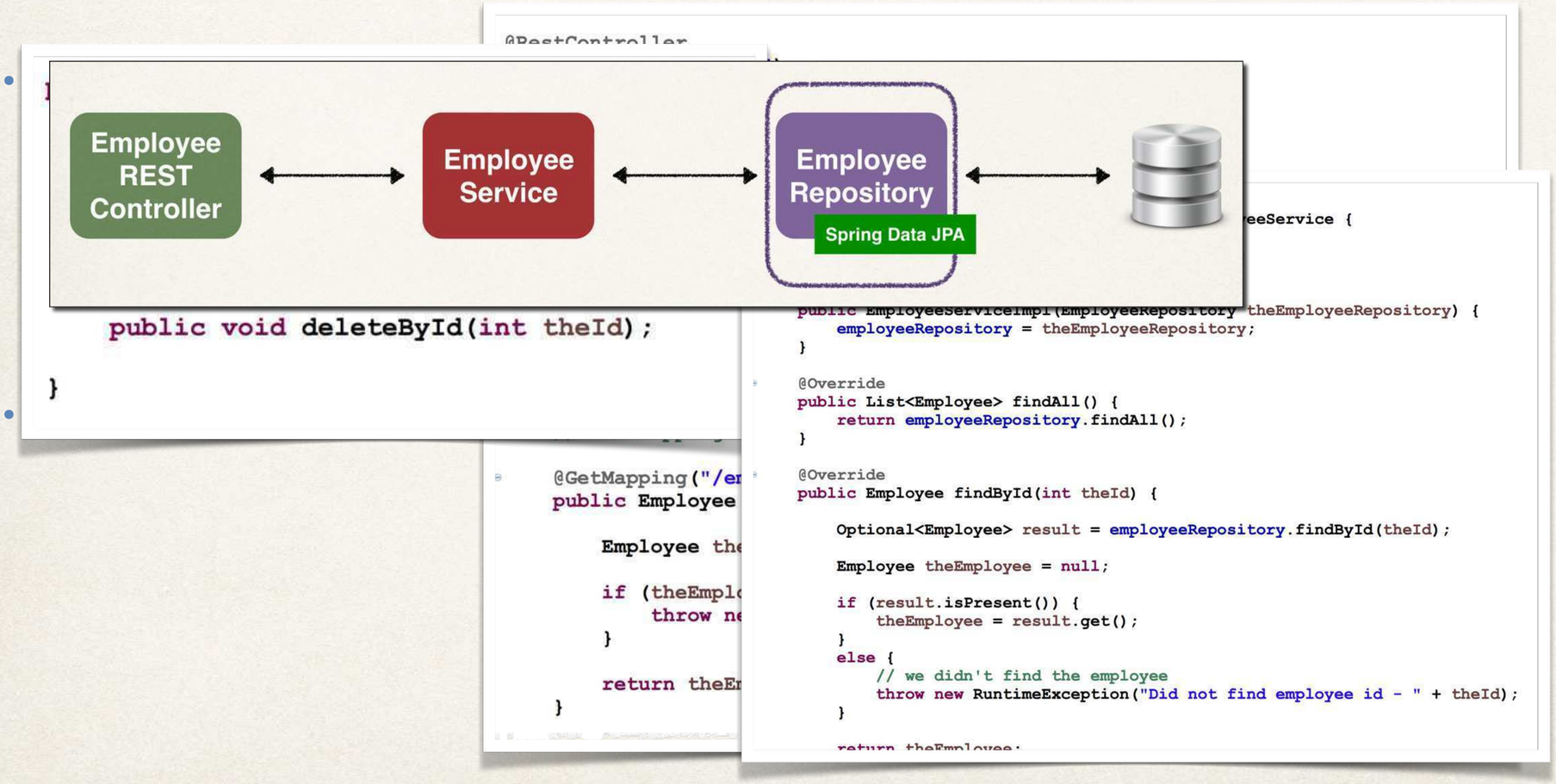
- Earlier, we saw the magic of Spring Data JPA
- This helped to eliminate boilerplate code



Hmmm ...
Can this apply to REST APIs?

The Problem

- We saw how to create a REST API for **`Employee`**



My Wish

- I wish we could tell Spring:

Create a REST API for me

Use my existing JpaRepository (entity, primary key)

Give me all of the basic REST API CRUD features for free

Spring Data REST - Solution

- **Spring Data REST** is the solution!!!!

https://spring.io/projects/spring-data-rest

- Leverages your existing **`JpaRepository`**

- Spring will give you a REST CRUD implementation for FREE …. like MAGIC!!
  - Helps to minimize boiler-plate REST code!!!
  - No new coding required!!!

Spring Data REST - How Does It Work?

- Spring Data REST will scan your project for **`JpaRepository`**
- Expose REST APIs for each entity type for your **`JpaRepository`**

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}
```

REST Endpoints

- By default, Spring Data REST will create endpoints based on entity type
- Simple pluralized form
  - First character of Entity type is lowercase
  - Then just adds an "s" to the entity

More on plural forms in later video

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}
```

`/employees`

Development Process

Step-By-Step

1. Add Spring Data REST to your Maven POM file

That's it!!!

Absolutely NO CODING required

Step 1: Add Spring Data REST to POM file

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

That's it!!!

Absolutely NO CODING required

Spring Data REST will scan for JpaRepository

HTTP Method		CRUD Action
POST	/employees	Create a new employee
GET	/employees	Read a list of employees
GET	/employees/{employeeId}	Read a single employee
PUT	/employees/{employeeId}	Update an existing employee
DELETE	/employees/{employeeId}	Delete an existing employee

Get these REST endpoints for free

In A Nutshell

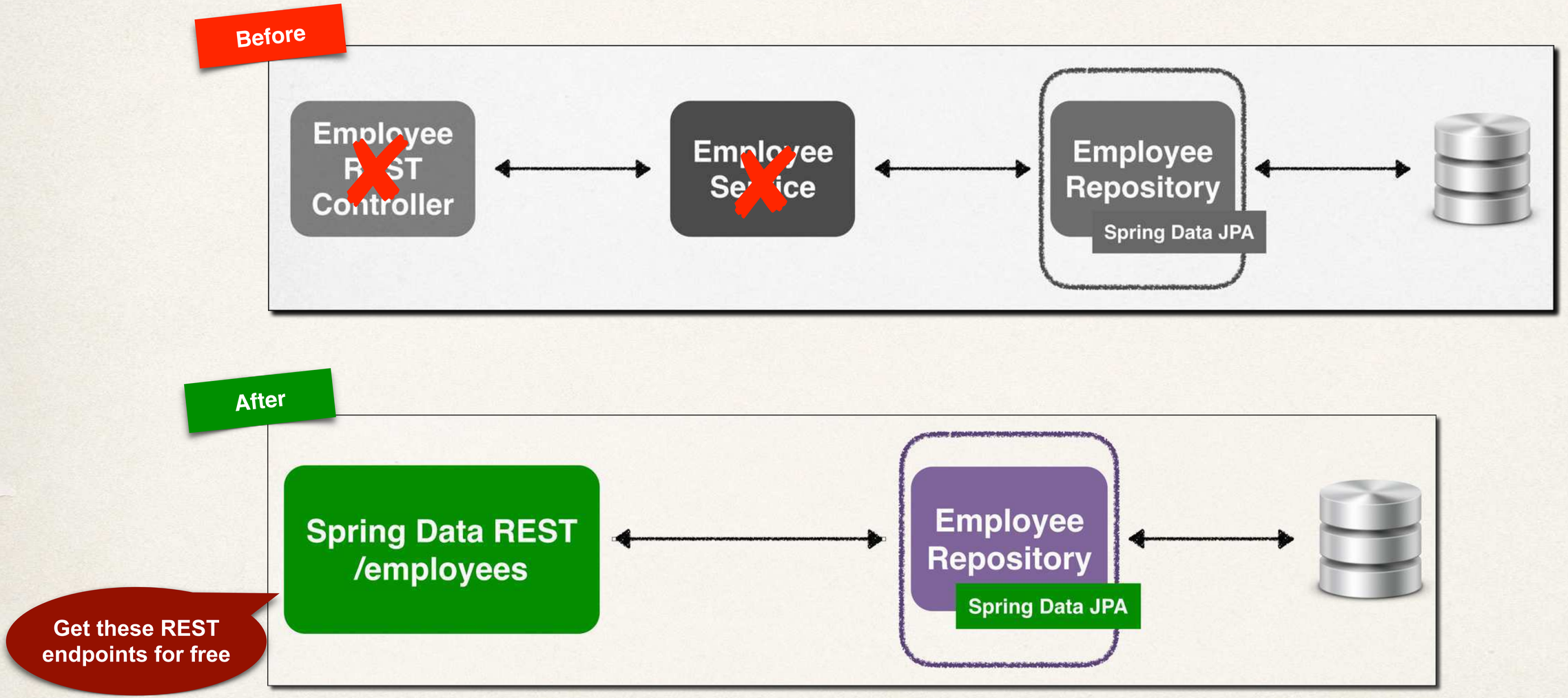
For Spring Data REST, you only need 3 items

1. Your entity: **`Employee`**
2. JpaRepository: **`EmployeeRepository extends JpaRepository`**
3. Maven POM dependency for: **`spring-boot-starter-data-rest`**

We already have these two

Only item that is new

Application Architecture



Minimized Boilerplate Code

Before Spring Data REST

After Spring Data REST

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

That's it!!!

Absolutely NO CODING required

Spring Data REST will scan for JpaRepository

HTTP Method		CRUD Action
POST	/employees	Create a new employee
GET	/employees	Read a list of employees
GET	/employees/{employeeId}	Read a single employee
PUT	/employees/{employeeId}	Update an existing employee
DELETE	/employees/{employeeId}	Delete an existing employee

Get these REST endpoints for free

HATEOAS

- Spring Data REST endpoints are HATEOAS compliant
  - **HATEOAS**: Hypermedia as the Engine of Application State

- Hypermedia-driven sites provide information to access REST interfaces
  - Think of it as meta-data for REST data

https://spring.io/projects/spring-hateoas

HATEOAS

- Spring Data REST response using HATEOAS
- For example REST response from: **`GET /employees/3`**

Get single employee

Response

```
{
    "firstName": "Avani",
    "lastName": "Gupta",
    "email": "avani@luv2code.com",
    "_links": {
        "self": {
            "href": "http://localhost:8080/employees/3"
        },
        "employee": {
            "href": "http://localhost:8080/employees/3"
        }
    }
}
```

Employee data

Response meta-data
Links to data

HATEOAS

- For a collection, meta-data includes page size, total elements, pages etc
- For example REST response from: **`GET /employees`**

Get list of employees

Response

```
{
    "_embedded": {
        "employees": [
            {
                "firstName": "Leslie",
                ...
            },
            ...
        ]
    },
    "page": {
        "size": 20,
        "totalElements": 5,
        "totalPages": 1,
        "number": 0
    }
}
```

JSON Array of employees

Response meta-data
Information about the page

More on configuring page sizes later

HATEOAS

- For details on HATEOAS, see

https://spring.io/projects/spring-hateoas

- HATEOAS uses Hypertext Application Language (HAL) data format
- For details on HAL, see

https://en.wikipedia.org/wiki/Hypertext_Application_Language

Advanced Features

- Spring Data REST advanced features
  - Pagination, sorting and searching
  - Extending and adding custom queries with JPQL
  - Query Domain Specific Language (Query DSL)

https://spring.io/projects/spring-data-rest

Spring Data REST
Configuration, Pagination and Sorting

REST Endpoints

- By default, Spring Data REST will create endpoints based on entity type
- Simple pluralized form
  - First character of Entity type is lowercase
  - Then just adds an "s" to the entity

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}
```

Pluralized Form

- Spring Data REST pluralized form is VERY simple
  - Just adds an "s" to the entity

- The English language is VERY complex!
  - Spring Data REST does NOT handle

Singular	Plural
Goose	Geese
Person	People
Syllabus	Syllabi
…	…

Problem

- Spring Data REST does not handle complex pluralized forms
  - In this case, you need to specify plural name

- What if we want to expose a different resource name?
  - Instead of **`/employees`** … use **`/members`**

Solution

- Specify plural name / path with an annotation

```
@RepositoryRestResource(path="members")
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}
```

http://localhost:8080/members

Pagination

- By default, Spring Data REST will return the first 20 elements
  - Page size = 20

- You can navigate to the different pages of data using query param

```
http://localhost:8080/employees?page=0

http://localhost:8080/employees?page=1

...
```

Spring Data REST Configuration

- Following properties available: application.properties

Name	Description
`spring.data.rest.base-path`	Base path used to expose repository resources
`spring.data.rest.default-page-size`	Default size of pages
`spring.data.rest.max-page-size`	Maximum size of pages
…	…

More properties available

www.luv2code.com/spring-boot-props

`spring.data.rest.*`

Sample Configuration

http://localhost:8080/magic-api/employees

File: application.properties

```
spring.data.rest.base-path=/magic-api

spring.data.rest.default-page-size=50
```

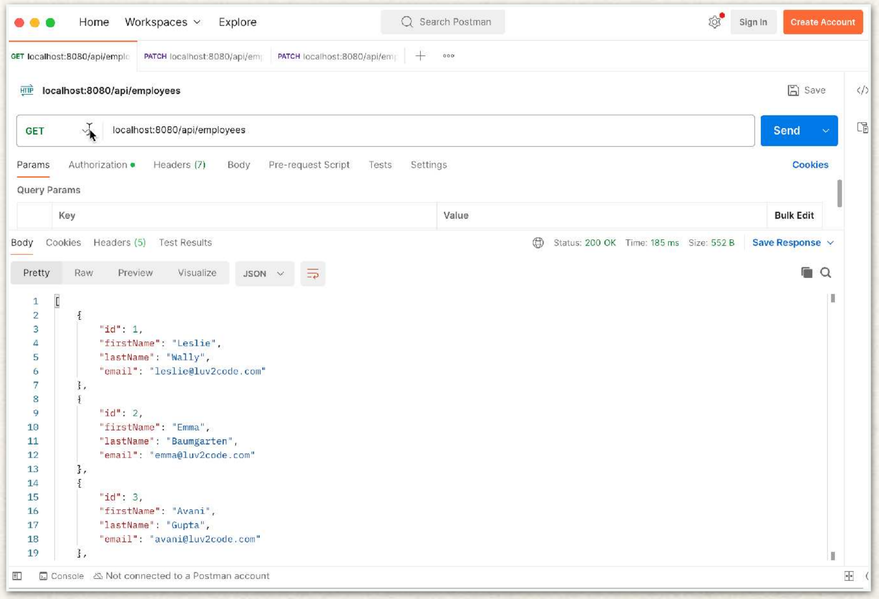
Returns 50 elements per page

Documenting REST APIs with OpenAPI and Swagger

The Problem

- There is a REST API out there … but we don't have any documentation
- We have to review the source code to find endpoints: @GetMapping etc
- Then use Postman or curl to call the REST API

```java
public class EmployeeRestController {

    // expose "/employees" and return a list of employees
    @GetMapping("/employees")  no usages
    public List<Employee> findAll() { return employeeService.findAll(); }

    // add mapping for GET /employees/{employeeId}

    @GetMapping("/employees/{employeeId}")  no usages
    public Employee getEmployee(@PathVariable int employeeId) {

        Employee theEmployee = employeeService.findById(employeeId);

        if (theEmployee == null) {
            throw new RuntimeException("Employee id not found - " + employeeId);
        }

        return theEmployee;
    }
```



My Wish

- I wish we could tell our app:

At run-time, generate API documentation

Inspect API endpoints based on Spring Configs, annotations etc

Provide a web UI for accessing endpoints No need for Postman

Springdoc To The Rescue

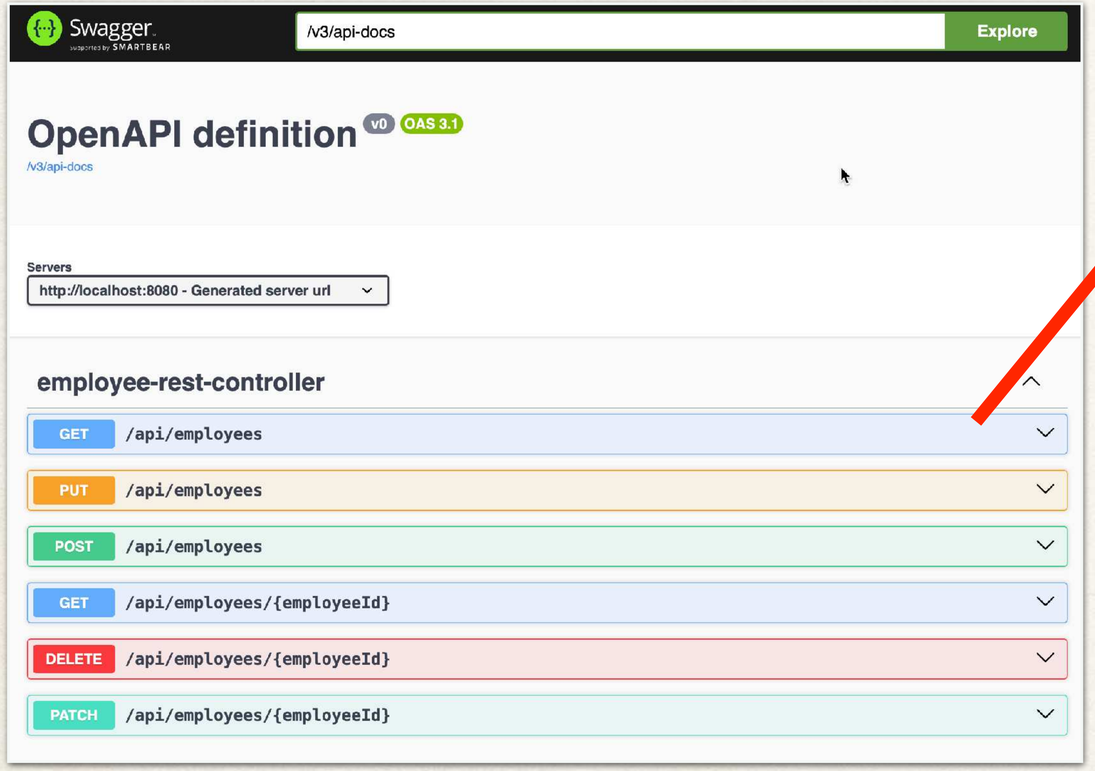
www.springdoc.org

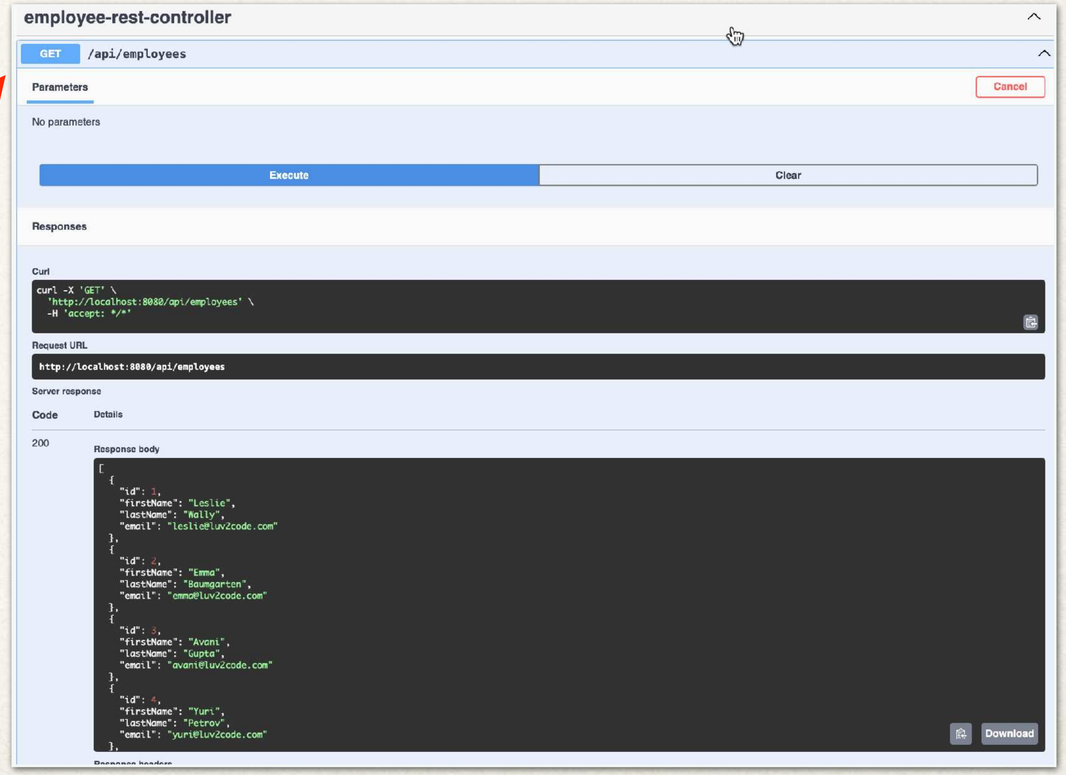
- **Springdoc** is separate open-source project
- Generates API documentation
- Inspects API endpoints based on Spring Configs, annotations etc
- Provides a web UI for accessing endpoints
  - No need for Postman

Springdoc - Swagger Web UI

www.springdoc.org

- Springdoc provides a **Swagger** web UI for accessing endpoints





Documenting REST APIs

www.springdoc.org

- OpenAPI is an industry standard format for documenting APIs
  - www.openapis.org

- **Swagger UI** is a browser-based UI for interacting with your API
  - Powered by Springdoc-OpenAPI

Development Process

1. Add Maven dependency for Springdoc

2. Access Swagger UI

3. Retrieve API endpoints as JSON or YAML

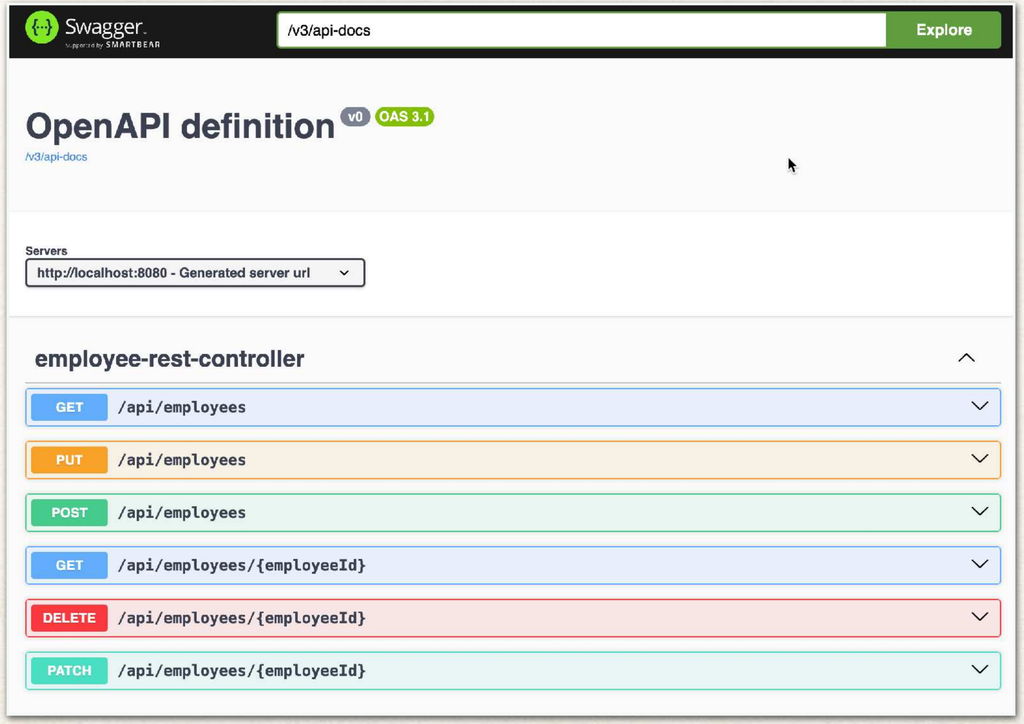
Step 1: Add Maven dependency for Springdoc

```
<dependency>
   <groupId>org.springdoc</groupId>
   <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
   <version>x.y.z</version>
</dependency>
```

www.springdoc.org

Step 2: Access the Swagger UI

- By default, Swagger UI is available at:
  - http://localhost:8080/swagger-ui/index.html

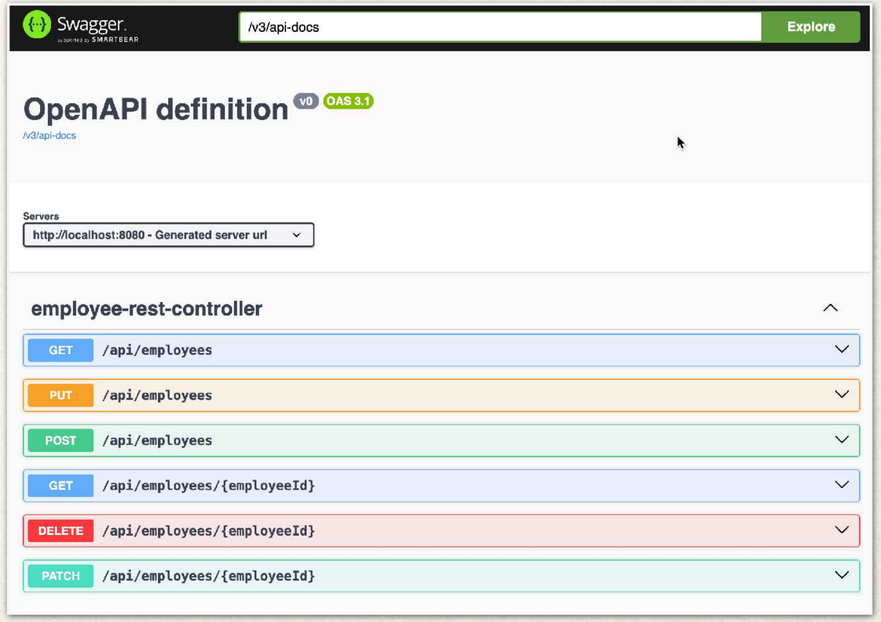


Configure Custom Path for Swagger UI

- Can configure a custom path in application.properties

```
# configure custom path
springdoc.swagger-ui.path=/my-fun-ui.html
```

- Access Swagger UI at
  - http://localhost:8080/my-fun-ui.html



Step 3: Retrieve API endpoints as JSON or YAML

- Docs for API endpoints available as JSON or YAML
- Useful for integration with other development tools
- Client SDK generation, API mocking, contract testing, etc
- JSON or YAML is language independent
- Can be processed by Python, Javascript, Go, C# etc

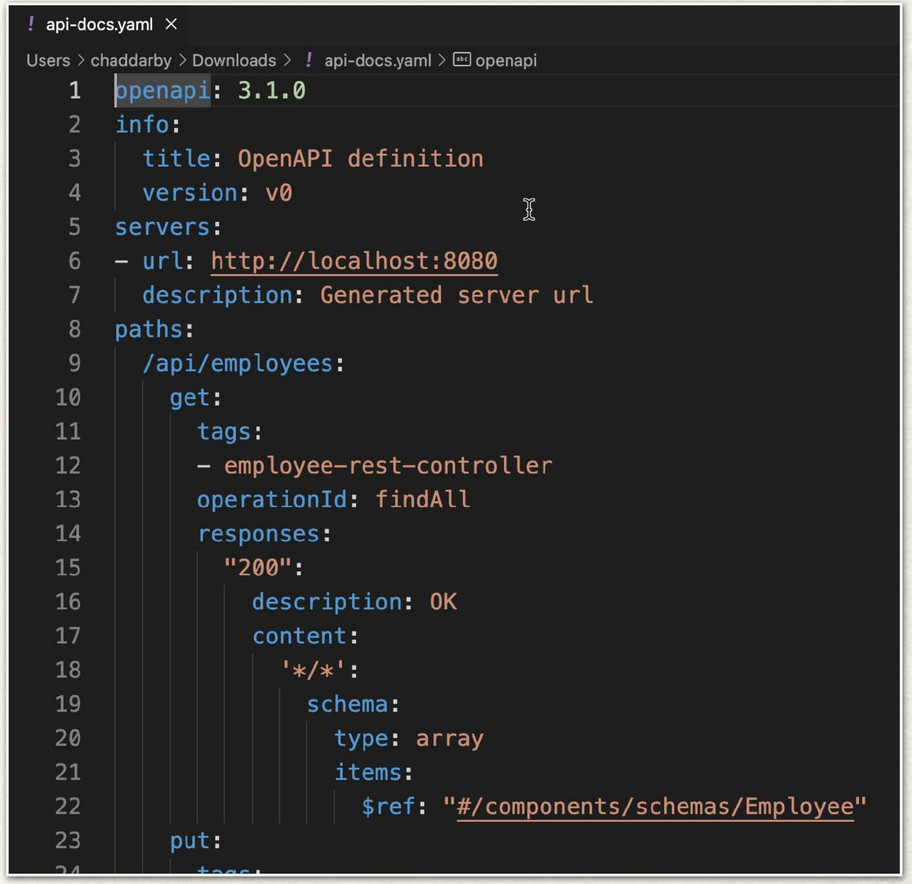
Step 3: Retrieve API endpoints as JSON or YAML

- By default, **JSON** docs available here
  - http://localhost:8080/v3/api-docs

Step 3: Retrieve API endpoints as JSON or YAML

- **YAML** docs available here
  - http://localhost:8080/v3/api-docs.yaml

- Web browser will download the YAML file
- You can view it with any text editor



Configure Custom Path for API docs

- Can configure a custom path in application.properties

```
# configure custom path
springdoc.api-docs.path=/my-api-docs
```

- Access API Docs at
  - http://localhost:8080/my-api-docs
  - http://localhost:8080/my-api-docs.yaml

JSON

YAML

Spring Data REST with OpenAPI and Swagger

Can I use OpenAPI and Swagger UI
with Spring Data REST???

Yes!
Use the same development process

Development Process

1. Add Maven dependency for Springdoc

2. Access Swagger UI

3. Retrieve API endpoints as JSON or YAML

Sorting

- You can sort by the property names of your entity
  - In our Employee example, we have: **`firstName`**, **`lastName`** and **`email`**

- Sort by last name (ascending is default)

```
http://localhost:8080/employees?sort=lastName
```

- Sort by first name, descending

```
http://localhost:8080/employees?sort=firstName,desc
```

- Sort by last name, then first name, ascending

```
http://localhost:8080/employees?sort=lastName,firstName,asc
```